Improving security of LSB Image Steganography by updating Pixel Locator Sequence

Dissertation submitted to
Visvesvaraya National Institute of Technology, Nagpur
in partial fulfillment of the requirements for the award of
the degree

Master of Technology
in
Computer Science & Engineering

by

Satyam Kumar (MT23MCS019)

under the guidance of

Dr. Swapan Maiti

Department of Computer Science & Engineering

Visvesvaraya National Institute of Technology

Nagpur 440 010 (India)

June 2025

Improving security of LSB Image Steganography by updating Pixel Locator Sequence

Dissertation submitted to
Visvesvaraya National Institute of Technology, Nagpur
in partial fulfillment of the requirements for the award of
the degree

Master of Technology
in
Computer Science & Engineering

by

Satyam Kumar (MT23MCS019)

under the guidance of

Dr. Swapan Maiti

Department of Computer Science & Engineering

Visvesvaraya National Institute of Technology

Nagpur 440 010 (India)

June 2025

©Visvesvaraya National Institute of Technology 2025

Department of Computer Science & Engineering
Visvesvaraya National Institute of Technology, Nagpur

Declaration

I, **Satyam Kumar**, hereby declare that this dissertation titled **" Improving security of LSB Image Steganography by updating Pixel Locator Sequence "** is carried out by me in the Department of Computer Science & Engineering of Visvesvaraya National Institute of Technology, Nagpur. The work is original and has not been submitted earlier whole or in part for the award of any degree at this or any other Institution.

Satyam Kumar
Date: 16.06.2025

Certificate

This to certify that the dissertation titled **" Improving security of LSB Image Steganography by updating Pixel Locator Sequence "** is submitted by **Satyam Kumar** in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Computer Science & Engineering**, VNIT Nagpur. The work is comprehensive, complete, and fit for final evaluation.

16/6/25

Dr. Swapan Maiti
Assistant Professor
Department of Computer Science & Engineering
VNIT Nagpur

Dr. M. P. Kurhekar
Head, Department of Computer Science
& Engineering, VNIT Nagpur
Date:

विभाग प्रमुख
Head
संगणक विज्ञान एवं अभियांत्रिकी विभाग
Dept. of Computer Science & Engg.
वि.रा.प्रौ.सं. नागपुर-१० (भारत)
V.N.I.T., Nagpur-440010. (India)

Department of Computer Science & Engineering
Visvesvaraya National Institute of Technology, Nagpur

Non-Plagiarism Certificate

Certified that M. Tech Dissertation titled **" Improving security of LSB Image Steganography by updating Pixel Locator Sequence "** submitted by **Satyam Kumar**, Enrollment No. **MT23MCS019**, ID No. **30702** has been checked for plagiarism using TURNITIN software and the overlap is found within prescribed limits. A summary of the report is given below.

Satyam_Thesis.pdf

ORIGINALITY REPORT

13%	10%	9%	5%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

Satyam Kumar
Enrollment No: MT23MCS019

16/6/25

Dr. Swapan Maiti
Assistant Professor
Department of Computer Science & Engineering
VNIT, Nagpur

ACKNOWLEDGEMENT

I would like to express my deepest gratitude and sincere appreciation to all those who have supported and guided me throughout this M.Tech journey.

First and foremost, I am immensely grateful to my supervisor, **Dr. Swapan Maiti** Sir, for his invaluable guidance, expertise, and encouragement. His constant support, patience, and insightful feedback have been instrumental in shaping this thesis. I am truly fortunate to have the opportunity to work under his mentorship.

I would like to extend my heartfelt thanks to **Dr. M. P. Kurhekar**, Head of Department, for his invaluable support and guidance throughout this journey. His encouragement has been vital in the successful completion of this project.

I would like to extend my heartfelt thanks to the faculty members of the Department of Computer Science & Engineering at Visvesvaraya National Institute of Technology Nagpur for their knowledge-sharing and for providing a conducive academic environment. I am indebted to them for their teachings and guidance which have broadened my perspective and enhanced my understanding of the subject. My sincere appreciation goes to them for their assistance, collaboration, and providing necessary resources for conducting experiments and carrying out this research work. Their contributions have been significant in the successful completion of this thesis.

I would also like to express my gratitude to my family and friends for their unwavering support, understanding, and encouragement throughout this journey. Their love, motivation, and belief in my abilities have been the driving force behind my accomplishments.

ABSTRACT

Image steganography enables covert communication by embedding data within digital images. Least Significant Bit (LSB) methods are widely used but vulnerable to steganalysis through predictable pixel modification patterns. To overcome this, we enhance the security of LSB image steganography using a cryptographically secure pseudo-random number gen- erator (CSPRNG) for unpredictable Pixel Locator Sequence (PLS) generation and derive keys with Elliptic Curve Diffie- Hellman (ECDH) key exchange and HMAC-based Key Derivation Function (HKDF)-derived AES encryption (providing forward secrecy). The message payloads and PLS data are encrypted using AES. The steganalysis is done using the StegExpose tool that analyzed stego images against Primary Sets, Chi-Square, Sample Pairs, and RS Analysis attacks, demonstrating 82% undetectability across 100 iStego100K images. This approach significantly advances LSB image steganography for secure covert communication.

TABLE OF CONTENTS

Title **Page No.**

ABSTRACT . . . . . **iv**

TABLE OF CONTENTS . . . . . **v**

LIST OF TABLES . . . . . **ix**

LIST OF FIGURES . . . . . **x**

1 INTRODUCTION **1**

1.1 Background and Motivation 1

1.2 Aim and objectives 3

1.2.1 Aim 3

1.2.2 Objectives 3

1.3 Thesis Outline 4

2 LITERATURE REVIEW **6**

2.1 Embedding Secrets in Pixels: Two-Level Security through DNA Encryption and LSB Image Steganography 6

2.1.1 Contribution 6

2.1.2 Key Points 6

2.1.3 Achievements 7

2.1.4 Drawbacks/Limitations 7

2.2 A Privacy-Preserving Scheme for Medical Diagnosis Records Using Encrypted Image Steganography 7

2.2.1 Contribution . . . 7
2.2.2 Key Points . . . 7
2.2.3 Achievements . . . 8
2.2.4 Drawbacks/Limitations . . . 8
2.3 Digital Image Steganography Survey and Investigation . . . 8
2.3.1 Contribution . . . 8
2.3.2 Key Findings . . . 8
2.3.3 Recent Developments . . . 9
2.3.4 Gaps Identified . . . 9
2.4 LSB Steganography Using Pixel Locator Sequence with AES . . . 9
2.4.1 Contribution . . . 9
2.4.2 Achievements . . . 9
2.4.3 Drawbacks/Limitations . . . 10
2.5 Summary . . . 10

3 METHODOLOGY . . . 12
3.1 Overview . . . 12
3.2 Research Approach . . . 12
3.3 Problem Definition and Scope . . . 14
3.4 System Overview . . . 16
3.4.1 System Description . . . 17
3.4.2 Block Diagram . . . 17
3.5 CSPRNG . . . 21
3.5.1 Introduction . . . 21
3.5.2 Key Properties . . . 22
3.5.3 Advantages over Other CSPRNG Architectures . . . 22
3.5.4 /dev/random and /dev/urandom . . . 24
3.6 Elliptic Curve Diffie-Hellman (ECDH) Key Exchange . . . 25
3.6.1 Introduction . . . 25
3.6.2 Mathematical Foundations . . . 25
3.6.3 ECDH Key Exchange Mechanism . . . 26

3.7 Advanced Encryption Standard (AES) 30
3.7.1 Introduction 30
3.7.2 Mathematical Foundations 31
3.7.3 AES Encryption Mechanism 32
3.8 XOR Operation 35
3.8.1 Introduction 35
3.8.2 Mathematical Foundations 36
3.9 Initialization Vector (IV) in AES 37
3.9.1 Introduction 37
3.9.2 Mathematical Foundations 37
3.9.3 IV in AES Encryption 38
3.9.4 Example: IV in AES-CBC Encryption 40
3.10 Least Significant Bit (LSB) Embedding 41
3.10.1 Introduction 41
3.10.2 Mathematical Foundations 42
3.10.3 Example: LSB Embedding 43
3.10.4 Discussion - Usage in Our Project 45
3.11 Pixel Locator Sequence (PLS) 46
3.12 Algorithms for LSB Steganography 46
3.12.1 Algorithm 1: PLS Generation 47
3.12.2 Algorithm 2: LSB Encoding 47
3.12.3 Algorithm 3: Decoding and Retrieval 49

4 IMPLEMENTATION 51
4.1 Overview 51
4.2 Tools and Technologies 51
4.2.1 Programming Languages 52
4.2.2 Libraries and Packages 52
4.3 Dataset Source 54
4.3.1 Sample Images - 5 Images 54
4.3.2 IStego100K Dataset 55

4.4 Implementation Environment 57
4.4.1 Development Environment Setup 57
4.4.2 File/Module Structure 58
4.4.3 Software or Hardware Specifications 59

5 EXPERIMENTAL RESULTS AND ANALYSIS 61
5.1 Overview 61
5.2 Evaluation Metrics 61
5.2.1 Visual Analysis 62
5.2.2 Statistical Analysis 63
5.2.3 StegExpose 64
5.2.4 Randomness Testing 66
5.3 Results and Interpretation 69
5.3.1 Visual Analysis Results 69
5.3.2 Statistical Analysis Results 70
5.3.3 Randomness Testing 74
5.3.4 Comparison with Baseline 75
5.4 Discussion 75
5.5 Limitations 76

6 CONCLUSION AND FUTURE WORK 78
6.1 Conclusion 78
6.2 Future Work 79

List of Tables

3.1 Comparison of CSPRNG Architectures 23

3.2 Cover Image Pixel Values 43

3.3 Stego-Image Pixel Values 44

5.1 StegExpose Command-Line Parameters 66

5.2 NIST Statistical Test Suite: Tests and Purposes 67

5.3 Evaluation of imperceptibility using PRNG for 5 images from the dataset used in [1]. 69

5.4 Evaluation of imperceptibility using CSPRNG for 5 images from the dataset used in [1]. 70

5.5 This table shows evaluation of imperceptibility using CSPRNG for 10 images from the Dataset IStego100K. 70

5.6 This table displays the steganalysis report for 5 images from the dataset used in [1] 71

5.7 This table displays the steganalysis report for 10 images from the Dataset IStego100K. 72

5.8 NIST Statistical Test Results for CSPRNG and PRNG 74

List of Figures

1.1 Document search results with the keyword “image steganography”. 2

3.1 Key Derivation Scheme 18
3.2 The Encoding Workflow of the System 19

4.1 Images 1(Left), 2(Middle), and 3(Right). 55
4.2 Images 4(Left) and 5(Right). 55
4.3 Images 1, 2, and 3 from IStego100K Dataset. 55
4.4 Images 4, 5, and 6 from IStego100K Dataset. 56
4.5 Images 7, 8, and 9 from IStego100K Dataset. 56
4.6 Image 10 from IStego100K Dataset. 56

Chapter 1

INTRODUCTION

1.1 Background and Motivation

In this digital communication age, confidentiality and integrity are paramount issues of concern. The volumes of information exchanged in recent years through unsecured networks have fostered the urgent demand for robust and sufficient solutions to assure sensitive information is secure from unauthorized access and malicious attacks. Information security is essential in safeguarding data, with two primary approaches: cryptography, which scrambles data, and data hiding, which conceals data within other media. Data hiding emphasizes imperceptibility, payload capacity, security, and robustness, and is categorized into:

- **Steganography:** Hides secret data within a cover medium.
- **Watermarking:** Embeds ownership information into media.

Steganography, derived from the Greek term for ”covered writing,” is the practice of concealing information within innocent mediums to evade detection. Unlike cryptography, which obscures the content of a message, steganography hides the very existence of the message, making it a critical tool in secure communications.

Steganography is a technique for hiding secret information within digital cover media, be it in the forms of images, audio, or video. Image-based methods are the most popular methods of steganography, due to the vast applications in secure communication.

LSB image steganography is a spatial domain method that hides secret data in the least significant bits of pixel values within a digital image. By changing these bits, which have only a small effect on how the image looks LSB steganography conceals information inside a cover image making a stego-image that looks almost the same as the original image.

Furthermore, research work on this topic is increasing every year, as evident from the search results on scopus.com for the last decade with the keyword ”image steganography”, see Figure 1.

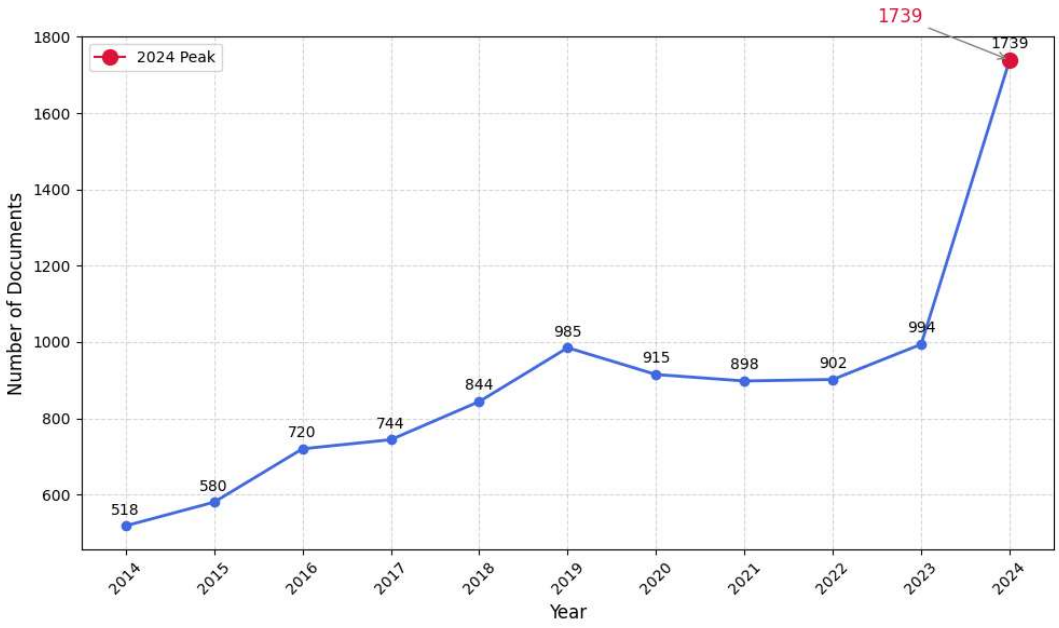

Figure 1.1: Document search results with the keyword “image steganography”.

The motivation for this thesis stems from the increasing relevance of steganography in modern cybersecurity and data protection. Digital images, due to their widespread use and complex structure, provide an ideal cover medium for concealing sensitive information. Among various image steganographic techniques, Least Significant Bit (LSB) insertion stands out for its simplicity and effectiveness in embedding data within the least significant bits of pixel values, rendering changes imperceptible to the human eye. However, LSB steganography is vulnerable to image manipulations and lossy compression, necessitating further research to enhance its robustness and security.

1.2 Aim and objectives

1.2.1 Aim

The primary aim of this project is to enhance the security and robustness of the LSB image steganography technique by addressing its inherent limitations in pixel selection and randomization. This involves the development of a more secure and efficient PLS generation mechanism, designed to resist statistical and visual attacks while preserving the visual quality of stego-images.

1.2.2 Objectives

To achieve the stated aim, the project will focus on:

1. **Strengthen Pixel Selection Security:** Replace conventional pseudo-random number generators with CSPRNGs to ensure high entropy and unpredictability in PLS generation.

2. **Improve Resistance to Attacks:** Incorporate obfuscation techniques, such as XOR operations, to obscure PLS patterns and prevent adversarial exploitation.

3. **Maintain Image Integrity:** Optimize the encoding algorithm to ensure that the visual quality of the cover image is minimally impacted, as measured by metrics such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM).

4. **Achieve Computational Efficiency:** Design and implement a secure LSB steganography method that is practical for real-world applications, maintaining low computational overhead.

5. **Support Scalability:** Develop solutions that can efficiently handle high-resolution images and larger datasets without compromising security or quality.

These objectives aim to provide a significant advancement in the domain of secure image steganography by creating a system that balances imperceptibility, robustness, and computational feasibility.

1.3 Thesis Outline

This thesis is organized into six chapters, each addressing a key aspect of the research work. The structure is designed to guide the reader logically from foundational concepts to implementation and analysis. The chapters are as follows:

1. **Chapter 1 – Introduction:**

   This chapter introduces the concept of steganography, with a focus on image-based and spatial domain techniques. It discusses the motivation for the study, defines the aim and objectives, and establishes the scope and significance of the research.

2. **Chapter 2 – Literature Review:**

   This chapter provides a critical review of existing literature in the field of image steganography. It analyzes various methods and techniques proposed in previous studies, highlighting their contributions, achievements, limitations, and the research gaps that this thesis aims to address.

3. **Chapter 3 – Methodology:**

   This chapter presents the overall research methodology, including problem definition, algorithm design, and the tools and technologies employed. It introduces the proposed techniques and elaborates on the underlying cryptographic and steganographic principles.

4. **Chapter 4 – System Design and Implementation:**

   This chapter describes the architecture and components of the proposed system. It details the implementation process, including the structure of scripts and modules developed, integration of security features, and use of datasets for testing.

5. **Chapter 5 – Experimental Results and Analysis:**

   The performance of the proposed system is evaluated in this chapter. It presents results from image quality assessments, steganalysis resistance, and performance testing. The outcomes are discussed in terms of effectiveness, efficiency, and security.

6. **Chapter 6 – Conclusion and Future Work:**

   The final chapter summarizes the findings of the research and discusses its broader implications. It outlines the limitations encountered and proposes directions for future work to enhance and expand upon the current system.

Chapter 2

LITERATURE REVIEW

A literature review is a comprehensive survey of existing research, theories, and knowledge related to a specific topic or field of study. It involves identifying, analyzing, and synthesizing scholarly works, including journal articles, books, conference papers, and other credible sources, to provide context and background for a research project.

2.1 Embedding Secrets in Pixels: Two-Level Security through DNA Encryption and LSB Image Steganography

2.1.1 Contribution

The author [2] here introduced a two-level security model by combining DNA encryption and the LSB technique. The method encodes the secret message into a DNA sequence, which is then XORed with a DNA key generated from human genome data. This encrypted sequence is embedded into an image using LSB steganography, maintaining high visual quality.

2.1.2 Key Points

- **Input:** 512x512 cover image and up to 8KB secret message.
- **Output:** A stego-image with an encrypted DNA sequence.

- **Performance:** Achieved imperceptibility with MSE of 0.0004–0.0334 and SSIM ¿ 0.99. PSNR for 8KB data was 68.32 dB (Lena) and 68.92 dB (Baboon), demonstrating high image quality.

2.1.3 Achievements

- DNA encryption provides a keyspace of 10^{200} combinations, making brute-force attacks infeasible.
- High fidelity (**IF** = 0.9996), ensuring minimal quality loss.

2.1.4 Drawbacks/Limitations

- **Data Capacity:** Stego image quality degrades as the payload increases, with PSNR dropping from 81.76 dB (2KB) to 68.32 dB (8KB).
- **Key Management:** Requires secure sharing of the DNA-based encryption keys.
- **Vulnerability:** Susceptible to statistical steganalysis due to detectable pixel patterns.
- **Computational Overhead:** DNA encryption increases encoding time, making it less scalable for larger datasets.

2.2 A Privacy-Preserving Scheme for Medical Diagnosis Records Using Encrypted Image Steganography

2.2.1 Contribution

Researchers[3] proposed a hybrid approach combining AES-128 encryption with LSB steganography for secure transmission of medical data. The scheme employs Diffie-Hellman key exchange for secure key management, supporting both client-server and normal modes.

2.2.2 Key Points

- Uses PNG images as carriers.

- The system offers imperceptibility, robustness, and capacity, surpassing five existing tools (e.g., StegExpose, OurSecret).

2.2.3 Achievements

- **Speed:** Encryption takes 2 seconds; embedding and extraction are nearly instantaneous.
- **Security:** Strong encryption ensures robustness against detection tools.

2.2.4 Drawbacks/Limitations

- **Limited Formats:** Only supports PNG/JPG formats.
- **Scalability Issues:** Capacity constrained by cover image size.
- **Steganalysis Resistance:** No evaluation against advanced steganalysis techniques.
- **Comparison Gaps:** Limited performance benchmarks against other hybrid systems.

2.3 Digital Image Steganography Survey and Investigation

2.3.1 Contribution

The author[4] conducted a comprehensive survey of image steganography methods, assessing their trade-offs between imperceptibility, payload capacity, security, and robustness. The study reviewed 97 works from 2015 onward, categorizing them into spatial and transform domain techniques.

2.3.2 Key Findings

- Spatial domain methods (e.g., LSB, PVD) are more prevalent but less robust compared to transform domain methods (e.g., DCT, DWT).

- Metrics like PSNR, SSIM, and MSE are standard for evaluating imperceptibility, while robustness is measured using Bit Error Rate (BER) and Character Error Ratio (CER).

2.3.3 Recent Developments

- Emerging techniques include GAN-based steganography and multi-channel embedding strategies.
- Popular datasets: USC-SIPI, BOSSbase, and UCID.

2.3.4 Gaps Identified

- Most methods focus on 1-2 goals, with limited solutions addressing all four aspects comprehensively.

2.4 LSB Steganography Using Pixel Locator Sequence with AES

2.4.1 Contribution

The paper proposes an enhancement to the traditional LSB steganography technique by incorporating a PLS combined with AES-256 encryption[1]. The PLS scatters data randomly across image pixels, making the location of embedded information harder to detect. AES encryption is applied not only to the data being embedded but also to the PLS, which further strengthens the security of the method. However, this approach requires that the encrypted PLS be sent along with the stego-image for decoding, resulting in a trade-off between security and space efficiency.

2.4.2 Achievements

- **Security:** The method offers strong defense against popular steganalysis tools like StegExpose. All test images passed the Primary Sets, Chi-Square, Sample Pairs, and

RS Analysis without detection, indicating robust resistance to traditional detection techniques.

- **Image Quality:** The method maintains high PSNR values (ranging from 44.57 dB to 48.35 dB) and low MSE (between 0.95 and 2.27), ensuring the image quality is preserved after data embedding.
- **Randomization:** The PLS method adds a layer of complexity by distributing the encrypted data across multiple random pixels (3 pixels per character, 9 RGB values), enhancing security by increasing the randomness and making it difficult for attackers to detect or predict the embedded data locations.

2.4.3 Drawbacks/Limitations

- **Algorithm 1 (rand() function):** The use of the rand() function for generating the PLS sequence is vulnerable to predictable patterns, potentially compromising the randomness and overall security of the system.
- **Algorithm 2 (PLS dependency):** The decoding process relies entirely on the PLS, meaning that if the PLS is lost or corrupted, the embedded data becomes irretrievable, introducing a single point of failure in the system.
- **Space Inefficiency:** The transmission of the encrypted PLS along with the stego-image increases the transmission size, reducing space efficiency, which could be problematic for bandwidth-constrained applications.
- **Computational Complexity:** The inclusion of AES encryption and the random pixel selection (via PLS) adds significant computational complexity, leading to slower encoding and decoding processes, which may be an issue in time-sensitive or resource-limited environments.

2.5 Summary

The literature review explores various approaches and advancements in the field of image steganography, focusing on techniques that enhance security, imperceptibility, and

robustness. [2] introduced a dual-layer security model combining DNA encryption and LSB steganography, achieving high image quality but facing challenges in scalability and statistical steganalysis resistance. [3] developed a hybrid AES-encrypted steganography scheme for medical data, excelling in speed and security but limited by format constraints and scalability. [4] provided a comprehensive survey categorizing spatial and transform domain methods, highlighting gaps in addressing all key goals simultaneously. [1] proposed an enhancement to LSB steganography using a PLS with AES encryption, offering robust security but encountering inefficiencies in space and computational performance.

Collectively, these studies underscore the trade-offs between capacity, security, and imperceptibility in steganographic techniques while identifying opportunities for further optimization and innovation.

Chapter 3

METHODOLOGY

3.1 Overview

The methodology of this project is structured to ensure a systematic and thorough approach to developing and evaluating a novel steganography algorithm. It begins with research approach, then an in-depth problem description, highlighting the challenges in securely embedding and extracting data within digital images while maintaining image quality and robustness against detection techniques. This is followed by a detailed concepts utilised, outlining the design and operational principles of the proposed method. In last the algorithm designed in this proposed method. This structured methodology ensures a holistic examination of the proposed solution and its practical applicability.

3.2 Research Approach

The research strategy for this study adopts a systematic approach to enhance the security of Least Significant Bit (LSB) image steganography by addressing vulnerabilities in Pixel Locator Sequence (PLS) generation and key management. The primary goal is to develop a secure steganographic system capable of covert communication, evaluated through rigorous experimentation and analysis.

Research Strategy Overview

This research follows an **experimental** approach, combined with a **proof-of-concept implementation**, to design and test an improved LSB steganography system. The experimental aspect involves developing a system that integrates a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) for PLS generation and an Elliptic Curve Diffie-Hellman (ECDH) key exchange with HKDF-derived AES keys for encryption. The proof-of-concept implementation demonstrates the feasibility of this system through a client-server architecture in Python, using the `cryptography` library for secure operations. The system is evaluated on a subset of 100 images from the iStego100K dataset, focusing on randomness, undetectability, and overall security.

Research Type: Quantitative

The research is primarily **quantitative**, relying on numerical metrics to assess the system's performance. Key evaluation methods include:

- **Visual Analysis(Image Quality)**: Peak Signal-to-Noise Ratio (PSNR), Mean Squared Error (MSE), and Structural Similarity Index (SSIM) are computed to evaluate imperceptibility.
- **Statistical Analysis(Steganalysis)**: In StegExpose, the Fusion score combines multiple statistical metrics (e.g., Primary Sets, Chi-Square, Sample Pairs, RS Analysis) to estimate the likelihood of hidden data in an image.
- **Randomness Testing**: The NIST Statistical Test Suite evaluates the randomness of the Pixel Locator Sequence (PLS) through p-values of various statistical tests, each targeting specific properties of a random sequence.

These metrics provide objective, measurable outcomes to validate the system's effectiveness, making a quantitative approach suitable.

Justification for the Approach

The experimental and proof-of-concept strategy is well-suited to the research goals of developing and validating a secure LSB steganography system. An experimental ap-

proach allows for controlled testing of the proposed enhancements replacing non-cryptographic PRNGs with a CSPRNG and transitioning to ECDH-based key management—ensuring that each component's impact on security and undetectability can be systematically assessed. The proof-of-concept implementation provides a practical demonstration of the system's feasibility, showing its applicability in real-world scenarios such as secure covert communication.

A quantitative methodology is appropriate because steganography research requires precise, objective evaluation of security and performance. Metrics like NIST test p-values and StegExpose Fusion scores offer concrete evidence of the system's improvements, directly addressing the vulnerabilities identified in traditional LSB methods. While qualitative insights could complement this work, the primary focus on undetectability and randomness necessitates a quantitative focus. This approach ensures that the research outcomes are rigorous, reproducible, and aligned with the objective of advancing secure steganographic techniques.

3.3 Problem Definition and Scope

This research addresses critical vulnerabilities in traditional Least Significant Bit (LSB) image steganography systems, particularly in the generation of the Pixel Locator Sequence (PLS) and key management. The primary problem lies in the use of insecure mechanisms that compromise the undetectability and confidentiality of hidden data, making such systems susceptible to statistical attacks and key compromise. By focusing on cryptographic randomness and secure key exchange, this study aims to develop a robust steganography system suitable for secure covert communication.

Problem Definition

Traditional LSB steganography systems often rely on the `rand()` function for generating the PLS, which determines the pixel locations for embedding secret data. However, the `rand()` function is not suitable for cryptographic applications due to several inherent limitations:

1. **Predictability of the `rand()` Function**:

    - **How `rand()` Works**: The `rand()` function is a basic pseudo-random number generator (PRNG) used in many programming languages, producing sequences based on an initial seed value.
    - **Why It's Predictable**: When the same seed is used, `rand()` generates identical sequences, enabling attackers to reverse-engineer or predict the values if the seed is known or guessed.
    - **Impact on PLS Generation**: Predictable PLS sequences expose the pixel locations containing sensitive data, allowing adversaries to detect and potentially extract the hidden message through statistical analysis, such as Chi-Square or RS Analysis.

2. **Limited Randomness**:

    - **Range Limitations**: The `rand()` function often has a limited output range (e.g., 0 to 32767 in some implementations), which is insufficient for cryptographic applications requiring large, uniform distributions across millions of pixels.
    - **Repetition**: Generated sequences may exhibit repetition over time, weakening randomization and enabling attackers to identify patterns in pixel selection.

3. **Non-Cryptographic Quality**:

    - **Weak Random Generation**: The `rand()` function lacks the strong randomness required for cryptographic purposes, as it is designed for general-purpose use rather than security.
    - **Vulnerability to Attacks**: The predictable patterns resulting from weak randomness provide clues about the embedded message, undermining the core principle of steganography—undetectability.

Additionally, traditional systems often employ password-based key derivation for encryption, which lacks forward secrecy. If a password is compromised, all past com-

munications are at risk, further exacerbating the security vulnerabilities introduced by an insecure PLS.

Research Focus: Steganography and Data Security

This study focuses on two critical aspects of steganography and data security:

- **Cryptographic Randomness for PLS Generation**: We aim to replace the non-cryptographic `rand()` function with a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG), specifically Python's `secrets.randbelow()`. This ensures that the PLS is unpredictable, reducing the likelihood of detectable patterns and enhancing resistance to statistical attacks. The Fisher-Yates shuffle algorithm, driven by CSPRNG outputs, further guarantees uniform pixel selection.
- **Secure Key Management with Forward Secrecy**: We transition from password-based key derivation to an Elliptic Curve Diffie-Hellman (ECDH) key exchange using the SECP256R1 curve, followed by HKDF to derive AES-CTR keys for message encryption and AES-CBC keys for PLS encryption. This approach introduces forward secrecy, ensuring that past communications remain secure even if a private key is compromised in the future.

By addressing these aspects, the research enhances the undetectability and confidentiality of LSB steganography, making it suitable for secure covert communication applications.

3.4 System Overview

This section provides a high-level description of the proposed enhanced LSB steganography system, which integrates cryptographic randomness and secure key management within a client-server architecture to address vulnerabilities in traditional methods. The system employs AES encryption with keys derived via ECDH key exchange, a cryptographically secure Pixel Locator Sequence (PLS) for pixel selection, and XOR obfuscation to enhance security. A block diagram illustrates the workflow, and the input, process,

and output flow are detailed through the encoding (client-side) and decoding (server-side) phases.

3.4.1 System Description

Conventional LSB steganography embeds data sequentially into a cover image's least significant bits, making it susceptible to statistical attacks due to predictable pixel selection patterns. Additionally, weak key management, such as password-based derivation, often lacks forward secrecy, exposing the hidden data to potential compromise. To address these vulnerabilities, this research proposes an improved LSB steganography system that leverages cryptographic techniques. The system operates within a client-server framework, where the client encodes a secret message into a cover image, and the server decodes it to retrieve the original message. Key management is facilitated through ECDH key exchange using the SECP256R1 curve, followed by HKDF to derive session-specific AES keys for encryption (AES-CTR for the message, AES-CBC for the PLS). The PLS is generated using Python's `secrets.randbelow()` and the Fisher-Yates shuffle, ensuring unpredictable pixel selection, and is further protected through XOR obfuscation.

3.4.2 Block Diagram

The system's workflow is illustrated in Figure 3.2, highlighting the encoding processes across the client and server. The diagram captures the flow from input (cover image and secret message) to output (stego image), including key steps such as key exchange, encryption, PLS generation, embedding, and extraction. The system's key derivation scheme is mentioned in Figure 3.1

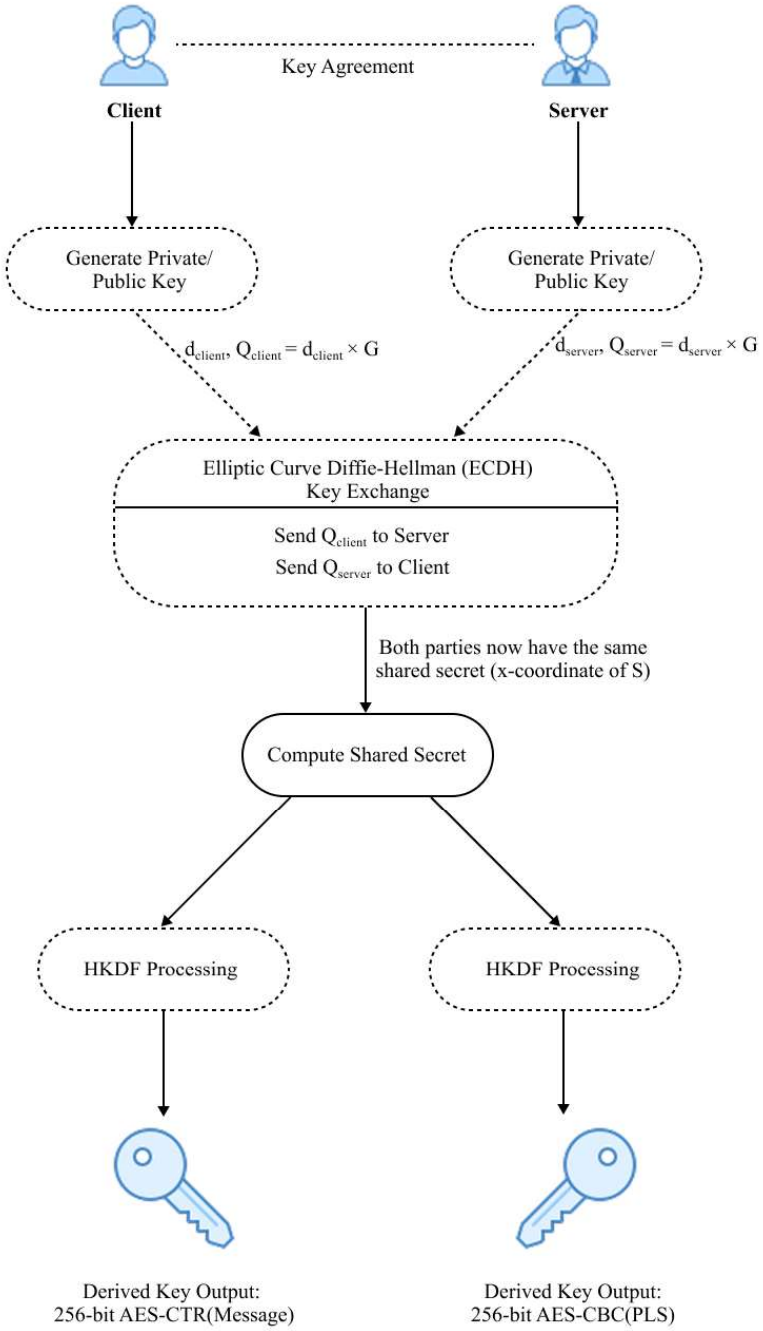

Figure 3.1: Key Derivation Scheme

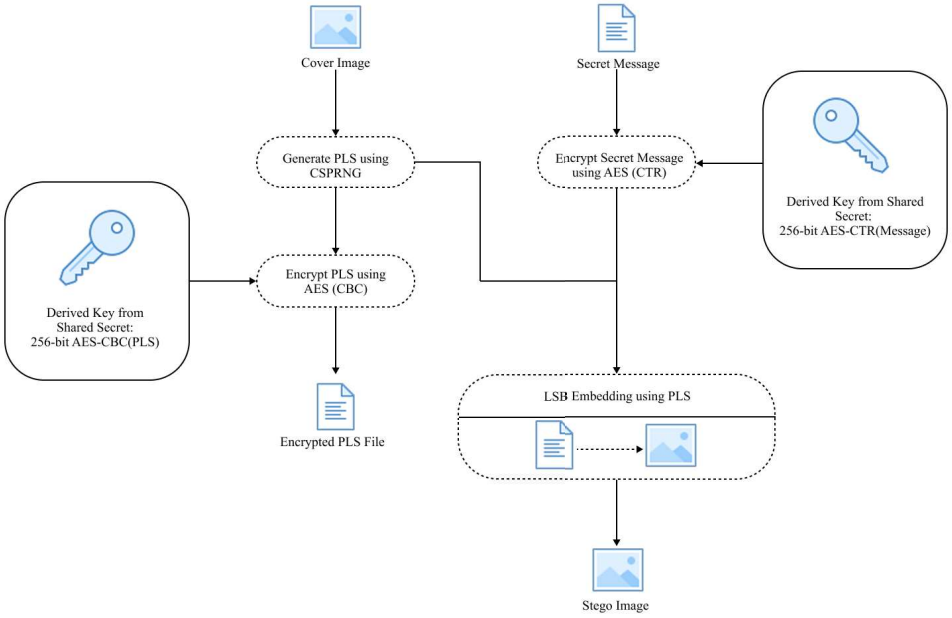
Cover Image
Secret Message
Generate PLS using CSPRNG
Encrypt Secret Message using AES (CTR)
Derived Key from Shared Secret: 256-bit AES-CTR(Message)
Derived Key from Shared Secret: 256-bit AES-CBC(PLS)
Encrypt PLS using AES (CBC)
Encrypted PLS File
LSB Embedding using PLS
Stego Image

Figure 3.2: The Encoding Workflow of the System

The workflow comprises three main phases: encoding (client-side), transmission, and decoding (server-side).

- **Encoding Workflow (Client)**:

  1. **Input**: A cover image (e.g., a 1024×1024 PNG from the iStego100K dataset) and a secret message to be hidden.
  2. **Key Exchange**: The client and server perform ECDH key exchange using the SECP256R1 curve to establish a shared secret. HKDF derives two 256-bit AES keys: one for message encryption (AES-CTR) and one for PLS encryption (AES-CBC).
  3. **Message Encryption**: The secret message is encrypted with AES-CTR using the derived message key, producing a ciphertext with a prepended initialization vector (IV).
  4. **PLS Generation**: A cryptographically secure PLS is generated using `secrets.randbelow()` and the Fisher-Yates shuffle, specifying random

pixel locations for embedding.

5. **LSB Embedding**: The encrypted message (including IV) is converted to binary, and each bit is embedded into the LSBs of RGB channels (3 bits per pixel) at PLS-specified locations, adjusting pixel parity (odd for 1, even for 0).
6. **PLS Obfuscation and Encryption**: The PLS is obfuscated via an XOR operation with a keystream derived from the shared secret, then encrypted with AES-CBC using the derived PLS key.
7. **Output**: The stego image and encrypted PLS are prepared for transmission.

- **Transmission**: The client transmits the stego image and encrypted PLS to the server over a secure channel.
- **Decoding Workflow (Server)**:

1. **Input**: The server receives the stego image and encrypted PLS.
2. **Key Exchange**: The server computes the same shared secret and derives the AES keys using ECDH and HKDF.
3. **PLS Decryption and Deobfuscation**: The encrypted PLS is decrypted with AES-CBC, then deobfuscated by reversing the XOR operation, recovering the original PLS.
4. **Message Extraction**: The encrypted message is extracted from the stego image's LSBs at PLS-specified locations.
5. **Message Decryption**: The ciphertext is decrypted with AES-CTR using the message key, yielding the original message.
6. **Output**: The recovered original message.

This architecture ensures both the message and its embedding locations are cryptographically protected, with forward secrecy preventing past session compromise. The block diagram in Figure 3.2 provides a visual summary of these processes.

3.5 CSPRNG

3.5.1 Introduction

A Cryptographically Secure PseudoRandom Number Generator (CSPRNG) is designed to produce sequences that are computationally indistinguishable from true random sequences, making them suitable for cryptographic applications such as key generation and nonce creation. The core design principle of the our system's CSPRNG, which is stream cipher-based using ChaCha20 and modeled after the architecture of `/dev/urandom`, centers on leveraging a fast and secure stream cipher to generate a keystream that serves as the random number output. ChaCha20, developed by Daniel J. Bernstein[5], is a modern stream cipher renowned for its robust security and high performance. The CSPRNG is initialized with a high-entropy seed derived from an entropy pool, which aggregates randomness from unpredictable sources such as hardware events (e.g., keyboard timings, mouse movements) and system noise (e.g., network packet timings). The generator operates in a non-blocking manner, ensuring continuous production of random numbers even when the entropy pool is depleted, by reusing the CSPRNG state until new entropy is available.

In the Linux kernel's `/dev/urandom`, which this CSPRNG emulates, ChaCha20 generates random numbers, with the entropy pool providing the initial seed and periodic reseeding to maintain long-term security. The security of the CSPRNG relies on computational indistinguishability, ensuring that no probabilistic polynomial-time algorithm can distinguish its output from a truly random sequence with non-negligible probability. Entropy, a critical measure of randomness, is quantified using Shannon's entropy formula:

$$H(X) = -\sum_{x} P(x) \log_2 P(x),$$

where $P(x)$ represents the probability of each possible value x. For cryptographic applications, seeds typically require at least 128 bits of entropy to ensure sufficient randomness.

3.5.2 Key Properties

The key properties of this stream cipher-based CSPRNG are as follows:

1. **Unpredictability**: The output of ChaCha20 is designed to be indistinguishable from true randomness, provided the key and nonce are secret and unique for each generation session. This ensures that predicting future outputs is computationally infeasible, even with knowledge of the algorithm and previous outputs.

2. **High Entropy**: The initial seed and periodic reseeding with fresh entropy from the pool maintain a high level of randomness, quantified by Shannon's entropy formula, ensuring robust cryptographic security.

3. **Speed**: ChaCha20 is highly efficient, capable of generating large volumes of random data quickly, making it ideal for applications requiring extensive random number generation.

4. **Non-blocking Operation**: Unlike `/dev/random`, this CSPRNG does not block when entropy is low, ensuring uninterrupted operation by reusing the CSPRNG state.

5. **Forward Secrecy**: With a large state space (256-bit key and 64-bit nonce) and proper nonce management, ChaCha20 ensures that compromising the current state does not facilitate prediction of future outputs.

These properties collectively ensure the CSPRNG's suitability for cryptographic applications, balancing security and performance.

3.5.3 Advantages over Other CSPRNG Architectures

Stream cipher-based CSPRNGs, such as those using ChaCha20, offer several advantages over other common CSPRNG architectures, including hash-based (e.g., Hash_DRBG) and block cipher-based (e.g., CTR_DRBG) designs. The following table summarizes these advantages:

- **Higher Speed**: Stream ciphers like ChaCha20 are generally faster than hash functions or block ciphers in software implementations, as they generate output incre-

Table 3.1: Comparison of CSPRNG Architectures

Feature	**Stream Cipher (ChaCha20)**	**Hash-Based (Hash_DRBG)**	**Block Cipher (CTR_DRBG)**
Speed	High (incremental output)	Moderate (block processing)	High (parallelizable)
Latency	Low (on-demand generation)	Higher (block-based)	Moderate (block-based)
Complexity	Simple (fewer components)	Moderate (hash iterations)	Complex (key/nonce management)
Patent Status	Patent-free	Patent-free	May involve patented ciphers
Standardization	Widely adopted (e.g., TLS)	Highly standardized (NIST)	Highly standardized (NIST)

mentally without requiring block processing. This efficiency is critical for applications needing large volumes of random data.

- **Lower Latency**: ChaCha20 produces output bit by bit or byte by byte, enabling on-demand generation with minimal latency, unlike hash-based or block cipher-based designs that process data in fixed-size blocks.

- **Simplicity**: The streamlined design of stream ciphers involves fewer components, reducing the risk of implementation errors and facilitating easier verification compared to the more complex key and nonce management in block cipher-based CSPRNGs.

- **Patent-Free**: ChaCha20 is not encumbered by patents, making it freely available for use in both open-source and proprietary software, unlike some block ciphers that may have licensing restrictions.

While hash-based CSPRNGs benefit from strong standardization (e.g., NIST SP 800-90A[6]) and block cipher-based CSPRNGs offer robust security in certain contexts, stream cipher-based CSPRNGs like ChaCha20 are often preferred for their performance and flexibility in software applications, particularly in modern protocols like TLS 1.3.

3.5.4 /dev/random and /dev/urandom

In the Linux operating system, `/dev/random` and `/dev/urandom` are special files providing access to the kernel's random number generator[7]. The `/dev/random` interface delivers random numbers directly from the entropy pool, which aggregates randomness from environmental noise, such as device drivers and user activities. If the entropy pool lacks sufficient randomness, reads from `/dev/random` block until more entropy is available, ensuring high-entropy output suitable for applications like generating long-term cryptographic keys.

In contrast, `/dev/urandom` employs a CSPRNG, specifically ChaCha20 in modern Linux kernels[5], seeded with the entropy pool. Unlike `/dev/random`, `/dev/urandom` does not block when entropy is low, instead reusing the CSPRNG state and periodically reseeding with new entropy. This non-blocking behavior makes `/dev/urandom` more

practical for most applications, providing a continuous supply of random numbers without delaying execution.

The user's CSPRNG, modeled after `/dev/urandom`, adopts this non-blocking approach, utilizing ChaCha20 to ensure both performance and cryptographic security. This design is advantageous in scenarios requiring a steady stream of random numbers, where blocking would impair application performance.

3.6 Elliptic Curve Diffie-Hellman (ECDH) Key Exchange

3.6.1 Introduction

Elliptic Curve Diffie-Hellman (ECDH) is a cryptographic key exchange protocol that enables two parties to establish a shared secret over an insecure channel. It leverages the algebraic properties of elliptic curves over finite fields, with security rooted in the computational hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP). ECDH is widely used in secure communication protocols, such as Transport Layer Security (TLS), Virtual Private Networks (VPNs), and secure file transfer systems, due to its efficiency and strong security with smaller key sizes compared to traditional Diffie-Hellman (DH) based on modular exponentiation. In the context of a client-server application (e.g., a secure file transfer system using Python's cryptographic libraries), ECDH is employed to derive a shared secret for encrypting messages and files, often using the NIST-standardized curve secp256r1 (P-256). This section provides a detailed explanation of ECDH, focusing on its mathematical foundations, operational mechanism using secp256r1, and a simplified example to demonstrate the underlying mathematics, suitable for a thesis submission.

3.6.2 Mathematical Foundations

An elliptic curve over a finite field $\mathbb{F}_p$, where p is a prime, is defined by the Weierstrass equation:

$$y^2 = x^3 + ax + b \pmod{p},$$

where $a, b \in \mathbb{F}_p$ are constants satisfying $4a^3 + 27b^2 \neq 0 \pmod{p}$ to ensure a non-singular curve. The set of points $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ satisfying this equation, along with a point at

infinity $\mathcal{O}$, forms an abelian group under point addition.

The group operations are:

- **Point Addition**: For distinct points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, the line through P and Q intersects the curve at a third point R', and the reflection of R' over the x-axis gives $R = P + Q$. The slope is:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \pmod{p},$$

and the coordinates of $R = (x_3, y_3)$ are:

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1 \pmod{p}.$$

- **Point Doubling**: For a point $P = (x_1, y_1)$, the tangent line at P intersects the curve at a point R', and the reflection gives $2P$. The slope is:

$$\lambda = \frac{3x_1^2 + a}{2y_1} \pmod{p},$$

with coordinates:

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = \lambda(x_1 - x_3) - y_1 \pmod{p}.$$

- **Identity**: The point at infinity $\mathcal{O}$ satisfies $P + \mathcal{O} = P$.

Scalar multiplication is defined as repeated addition: for a point P and integer k, $kP = P + P + \cdots + P$ (k times), typically computed efficiently using the double-and-add algorithm. The security of ECDH relies on the ECDLP: given points G and $Q = kG$, computing k is computationally infeasible for large k and well-chosen curves.

3.6.3 ECDH Key Exchange Mechanism

In the context of the client-server application, ECDH is implemented using the secp256r1 curve, a 256-bit prime field curve standardized by NIST [8]. The key exchange proceeds as follows:

1. **Setup**: Both parties agree on the secp256r1 curve over $\mathbb{F}_p$, where p is a 256-bit prime, defined by parameters (p, a, b), a base point G of order n (a large prime), and the curve equation $y^2 = x^3 - 3x + b \pmod{p}$. The curve parameters are standardized to ensure interoperability.

2. **Private Key Selection**: Each party selects a private key, a random integer in $[1, n-1]$:

   - The client chooses d_A.
   - The server chooses d_B.

3. **Public Key Generation**: Each party computes their public key by scalar multiplication:

   - The client computes $Q_A = d_A G$.
   - The server computes $Q_B = d_B G$.

   Public keys are serialized into bytes (e.g., SEC1 compressed format, 33 bytes for secp256r1) for transmission.

4. **Key Exchange**: The parties exchange their public keys Q_A and Q_B over the insecure channel, typically using a socket connection with serialization via Python's `pickle`.

5. **Shared Secret Computation**: Each party computes the shared secret using their private key and the other's public key:

   - The client computes $S = d_A Q_B = d_A(d_B G)$.
   - The server computes $S = d_B Q_A = d_B(d_A G)$.

   Since $d_A(d_B G) = d_B(d_A G) = d_A d_B G$, both compute the same point S. The x-coordinate of S (a 256-bit integer) is used as the shared secret, often processed through a key derivation function (e.g., HKDF) to generate AES keys for encryption.

The security relies on the ECDLP: an eavesdropper observing G, Q_A, and Q_B cannot compute d_A, d_B, or S without solving the ECDLP, which is intractable for secp256r1's 256-bit field.

Example: ECDH with a Toy Curve

Since secp256r1 uses a 256-bit field, manual computation is impractical for a thesis example. Instead, we use a simplified elliptic curve over a small field to illustrate the mathematics, noting that the structure mirrors secp256r1's operations.

Curve Parameters:

- Field: $\mathbb{F}_{23}$, so $p = 23$.
- Equation: $y^2 = x^3 + x + 1 \pmod{23}$, where $a = 1$, $b = 1$.
- Base point: $G = (5, 1)$, with order $n = 7$ (a small prime for simplicity).

Step 1: Private Key Selection:

- Client chooses $d_A = 3$.
- Server chooses $d_B = 4$.

Step 2: Public Key Generation:

- Client computes $Q_A = d_A G = 3G$:
  - Compute $2G$: For $G = (5, 1)$, slope:

$$\lambda = \frac{3(5)^2 + 1}{2(1)} = \frac{75 + 1}{2} = 38 \equiv 15 \pmod{23}.$$

  - $x_3 = \lambda^2 - 2(5) = 15^2 - 10 = 225 - 10 = 215 \equiv 8 \pmod{23}$.
  - $y_3 = \lambda(5 - 8) - 1 = 15(-3) - 1 = -46 \equiv 0 \pmod{23}$.
  - So, $2G = (8, 0)$.
  - Compute $3G = 2G + G$: Slope:

$$\lambda = \frac{1 - 0}{5 - 8} = \frac{1}{-3} \equiv 1 \cdot 15 \pmod{23},$$

where $-3 \equiv 20 \pmod{23}$, $20^{-1} \equiv 15 \pmod{23}$.

- $x_3 = \lambda^2 - 8 - 5 = 15^2 - 13 = 225 - 13 = 212 \equiv 20 \pmod{23}$.
- $y_3 = \lambda(8 - 20) - 0 = 15(-12) = -180 \equiv 21 \pmod{23}$.
- So, $Q_A = 3G = (20, 21)$.

- Server computes $Q_B = d_B G = 4G$:
  - From above, $2G = (8, 0)$.
  - Compute $3G = (20, 21)$.
  - Compute $4G = 3G + G$: Slope:

$$\lambda = \frac{1 - 21}{5 - 20} = \frac{-20}{-15} = \frac{4}{3} \equiv 4 \cdot 8 \pmod{23},$$

    where $3^{-1} \equiv 8 \pmod{23}$, so $\lambda = 32 \equiv 9 \pmod{23}$.
  - $x_3 = \lambda^2 - 20 - 5 = 9^2 - 25 = 81 - 25 = 56 \equiv 10 \pmod{23}$.
  - $y_3 = \lambda(20 - 10) - 21 = 9 \cdot 10 - 21 = 90 - 21 = 69 \equiv 0 \pmod{23}$.
  - So, $Q_B = 4G = (10, 0)$.

Step 3: Key Exchange:

- Client sends $Q_A = (20, 21)$ to the server.
- Server sends $Q_B = (10, 0)$ to the client.

Step 4: Shared Secret Computation:

- Client computes $S = d_A Q_B = 3(10, 0)$:
  - Compute $2(10, 0)$: Slope $\lambda = \frac{3(10)^2+1}{2(0)} = \frac{301}{0}$, undefined, so $2(10, 0) = \mathcal{O}$.
  - Compute $3(10, 0) = \mathcal{O} + (10, 0) = (10, 0)$.
  - This indicates a need to compute $d_A d_B G$.
- Server computes $S = d_B Q_A = 4(20, 21)$:

- Compute $S = d_A d_B G = 3 \cdot 4G = 12G$. Since $n = 7$, $12 \equiv 5 \pmod 7$, so $S = 5G$.
- Compute $5G = 4G + G = (10, 0) + (5, 1)$:
- Slope: $\lambda = \frac{1-0}{5-10} = \frac{1}{-5} \equiv 1 \cdot 14 \pmod{23}$, where $-5 \equiv 18 \pmod{23}$, $18^{-1} \equiv 14 \pmod{23}$.
- $x_3 = \lambda^2 - 10 - 5 = 14^2 - 15 = 196 - 15 = 181 \equiv 20 \pmod{23}$.
- $y_3 = \lambda(10 - 20) - 0 = 14(-10) = -140 \equiv 2 \pmod{23}$.
- So, $S = 5G = (20, 2)$.

Result: Both compute $S = (20, 2)$, and the shared secret is the x-coordinate, 20. In practice, as in the client-server application, the x-coordinate is processed (e.g., via HKDF) to derive AES keys.

The client-server application uses ECDH with the secp256r1 curve to establish a shared secret for encrypting a secret message (embedded in an image via steganography) and a file. The choice of secp256r1 is appropriate due to its 128-bit security level, compatibility with TLS, and efficient implementation in libraries like Python's `cryptography`. The toy example over $\mathbb{F}_{23}$ simplifies the mathematics for clarity but is insecure due to its small parameters, which are vulnerable to brute-force attacks. In contrast, secp256r1's 256-bit field ensures the ECDLP is intractable, requiring approximately 2^{128} operations to break. The application serializes public keys into bytes, likely using SEC1 format, and derives AES keys from the shared secret's x-coordinate, demonstrating a practical use of ECDH in secure communication. ECDH's efficiency, with smaller keys compared to RSA or traditional DH, makes it ideal for resource-constrained environments.

3.7 Advanced Encryption Standard (AES)

3.7.1 Introduction

The Advanced Encryption Standard (AES) is a symmetric-key block cipher standardized by the National Institute of Standards and Technology (NIST) in 2001 [9]. It is widely used to secure sensitive data in applications such as Transport Layer Security (TLS), disk

encryption, and secure file transfers. AES operates on fixed-size blocks of 128 bits (16 bytes) and supports key sizes of 128, 192, or 256 bits, with AES-256 being the most secure variant. In the context of a client-server secure file transfer system, AES-256 is employed in Cipher Block Chaining (CBC) mode for file encryption and Counter (CTR) mode for text encryption, with keys derived from an Elliptic Curve Diffie-Hellman (ECDH) shared secret using the secp256r1 curve. This section provides a detailed explanation of AES, focusing on its mathematical foundations, operational mechanism as implemented in the system, and a simplified example to demonstrate the underlying mathematics, suitable for a thesis submission.

3.7.2 Mathematical Foundations

AES is a substitution-permutation network (SPN) that operates on a 128-bit block, represented as a 4×4 byte matrix (state) over the finite field GF(2^8). The field is defined by the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, and arithmetic operations (addition, multiplication, and inversion) are performed modulo this polynomial. Each round of AES involves transformations that provide confusion (non-linearity) and diffusion (spreading input bits across the output).

The AES-256 algorithm consists of:

- **Key Expansion**: Generates 15 round keys (each 128 bits) from the 256-bit key using a key schedule involving the Rijndael S-box, rotation, and XOR operations.
- **Initial Round**: Adds the first round key to the state using bitwise XOR.
- **Main Rounds** (13 for AES-256): Each round applies:
  - **SubBytes**: Substitutes each byte using a non-linear S-box based on inversion in GF(2^8).
  - **ShiftRows**: Cyclically shifts the rows of the state matrix (row i shifts left by i bytes).
  - **MixColumns**: Multiplies each column by a fixed polynomial in GF(2^8), providing diffusion.

  - **AddRoundKey**: XORs the state with the round key.

- **Final Round** (14th): Omits MixColumns, applying only SubBytes, ShiftRows, and AddRoundKey.

The encryption process can be expressed as:

$$\text{Ciphertext} = \text{AES}_k(\text{Plaintext}) = \text{Round}_{14}(\text{Round}_{13}(\cdots \text{Round}_0(\text{Plaintext}) \cdots)),$$

where each round is a composition of the transformations, and k is the 256-bit key.

The project uses AES in two modes:

- **CBC Mode**: For file encryption, each block is XORed with the previous ciphertext block (or IV for the first block) before encryption, requiring PKCS7 padding to align data to the 16-byte block size.

- **CTR Mode**: For text encryption, converts AES into a stream cipher by encrypting a counter (initialized with an IV) to produce a keystream, XORed with the plaintext, eliminating the need for padding.

3.7.3 AES Encryption Mechanism

In the client-server system, AES-256 is implemented as follows, using the Python `cryptography` library:

1. **Key Derivation**: An ECDH key exchange (using secp256r1) produces a shared secret, processed by HKDF with SHA256 to derive two 256-bit keys:

   - `message_key`: For AES-CTR encryption of a secret message.
   - `pls_key`: For AES-CBC encryption of a file.

   The HKDF derivation is:

$$\text{key\_material} =$$

$$\text{HKDF}_{\text{SHA256}}(\text{shared\_secret}, \text{length} = 64, \text{salt} = \text{None}, \text{info} = \text{“steganography\_keys”}),$$

   where key_material$[:32]$ is `message_key` and key_material$[32:]$ is `pls_key`.

2. **Text Encryption (CTR Mode)**:

   - A 16-byte IV is generated using `os.urandom(16)`.
   - The message is encoded as UTF-8 and encrypted:

$$\text{Ciphertext} = \text{Plaintext} \oplus \text{AES}_k(\text{Counter}(\text{IV}, i)),$$

     where $\text{Counter}(\text{IV}, i)$ increments the counter for each block.
   - The IV and ciphertext are embedded in an image via steganography.

3. **File Encryption (CBC Mode)**:

   - A 16-byte IV is generated using `os.urandom(16)`.
   - The file data is padded with PKCS7 to ensure a multiple of 16 bytes.
   - Encryption proceeds as:

$$C_i = \text{AES}_k(P_i \oplus C_{i-1}), \quad C_0 = \text{IV},$$

     where P_i is the i-th plaintext block, and C_i is the i-th ciphertext block.
   - An HMAC-SHA256 is computed over the ciphertext using `pls_key` for integrity.
   - The output file contains: original file size (8 bytes), IV (16 bytes), ciphertext, and HMAC (32 bytes).

4. **Transmission**: The encrypted message (in the stego-image) and file are sent to the server, which uses the same ECDH-derived keys to decrypt.

The use of CBC for files ensures robust encryption with diffusion, while CTR for text allows flexible, padding-free encryption suitable for variable-length messages.

Example: AES Single-Block Encryption

To illustrate AES's mathematics, we encrypt a single 128-bit block using a simplified 128-bit key (AES-256 is too complex for manual computation). This example demon-

strates the core transformations for one round, noting that the project uses AES-256 with 14 rounds.

Parameters:

- Plaintext: $P =$ 0x3243f6a8885a308d313198a2e0370734 (16 bytes).
- Key: $K =$ 0x2b7e151628aed2a6abf7158809cf4f3c (16 bytes, for AES-128).
- S-box: A simplified lookup (e.g., first byte $0x32 \rightarrow 0x63$).

Step 1: State Representation:

$$\text{State} = \begin{bmatrix} 32 & 88 & 31 & e0 \\ 43 & 5a & 31 & 37 \\ f6 & 30 & 98 & 07 \\ a8 & 8d & a2 & 34 \end{bmatrix} \text{(hex bytes)}.$$

Step 2: AddRoundKey (Initial Round):

$$\text{State} \leftarrow \text{State} \oplus K_0,$$

where $K_0 = K$. For simplicity, assume $K_0 =$ State, so XOR yields a new state (in practice, computed byte-wise).

Step 3: SubBytes: Apply the S-box to each byte. Suppose $0x32 \rightarrow 0x63$, $0x43 \rightarrow 0x7c$, etc., yielding:

$$\text{State} \leftarrow \begin{bmatrix} 63 & \cdot & \cdot & \cdot \\ 7c & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

Step 4: ShiftRows: Shift row i left by i bytes:

$$\text{State} \leftarrow \begin{bmatrix} 63 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 7c \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

Step 5: MixColumns: Multiply each column by a fixed matrix in GF(2^8). For column 1, this is complex, so assume a partial result.

Step 6: AddRoundKey: XOR with the next round key (from key expansion, not shown).

Result: The ciphertext is the final state after transformations. In practice, 14 rounds are applied for AES-256, with the project's CBC and CTR modes chaining blocks or counters, respectively.

This example is simplified; real AES-256 uses a 256-bit key, 14 rounds, and mode-specific operations (CBC or CTR) as in the project.

The client-server system uses AES-256 in CBC mode for file encryption, ensuring diffusion and integrity via HMAC-SHA256, and in CTR mode for text encryption, allowing padding-free operation. The 256-bit keys, derived from an ECDH shared secret using HKDF, provide approximately 128 bits of security, suitable for high-security applications. The use of random IVs (`os.urandom(16)`) ensures semantic security, preventing pattern attacks. CBC mode requires padding, handled by PKCS7, while CTR mode's stream-like behavior suits the variable-length secret message. The example simplifies AES to one block and round, but the project's implementation handles multi-block data efficiently, with HMAC ensuring authenticity for files. AES's efficiency and security make it ideal for the system's secure file transfer and steganography-based communication.

3.8 XOR Operation

3.8.1 Introduction

The XOR (exclusive OR) operation is a fundamental bitwise cryptographic technique widely used in encryption and obfuscation schemes due to its simplicity, reversibility, and efficiency. In the context of a client-server secure file transfer system employing Least Significant Bit (LSB) steganography, XOR is utilized to obfuscate the Pixel Locator Sequence (PLS), a sequence that specifies pixel locations for embedding a secret message in an image. The obfuscated PLS is subsequently encrypted with AES in Cipher Block Chaining (CBC) mode, enhancing security against statistical attacks. The XOR opera-

tion's keystream is derived from a 256-bit shared secret established via Elliptic Curve Diffie-Hellman (ECDH) using the secp256r1 curve, ensuring cryptographic randomness. This section provides a detailed examination of the XOR operation, its mathematical foundations, its mechanism within the steganography system, and a mathematical example to illustrate its application, suitable for a thesis submission.

3.8.2 Mathematical Foundations

The XOR operation is a binary operation defined on bits a and b:

$$a \oplus b = \begin{cases} 1 & \text{if } a \neq b, \\ 0 & \text{if } a = b. \end{cases} \tag{3.1}$$

For two bit sequences $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, the XOR operation is applied element-wise:

$$A \oplus B = (a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n). \tag{3.2}$$

Key properties of XOR include:

- **Reversibility**: $(A \oplus B) \oplus B = A$, as $a \oplus b \oplus b = a$.
- **Commutativity**: $A \oplus B = B \oplus A$.
- **Associativity**: $(A \oplus B) \oplus C = A \oplus (B \oplus C)$.
- **Self-Inverse**: $A \oplus A = 0$.

When XOR is applied with a cryptographically secure keystream (e.g., generated by a CSPRNG or derived from a secure key), the output is statistically indistinguishable from random noise, provided the keystream is uniformly random and secret. This property is leveraged in stream ciphers and obfuscation schemes, as the XOR of a message with a random keystream preserves randomness.

In the system, the PLS, a binary sequence defining pixel locations, is obfuscated by XORing with a keystream derived from the ECDH shared secret, ensuring that the obfuscated sequence exhibits no detectable patterns.

The XOR operation enhances the security of the LSB steganography system by obfuscating the PLS, ensuring that pixel locations remain obscured even if an attacker intercepts the encrypted PLS file. The keystream, derived from the ECDH shared secret via `pls_key`, provides 2^{256} possible sequences, making brute-force attacks infeasible. The randomness of the keystream, likely generated by a CSPRNG or directly from `pls_key`, ensures that P' is statistically indistinguishable from noise, contributing to the system's 82% undetectability rate as validated by StegExpose analysis. The XOR operation's computational efficiency ($O(n)$) introduces minimal overhead, aligning with the system's performance goals. The layered approach—XOR obfuscation followed by AES-CBC encryption with HMAC-SHA256—strengthens resistance to statistical and tampering attacks.

3.9 Initialization Vector (IV) in AES

3.9.1 Introduction

The Initialization Vector (IV) is a pivotal element in block cipher modes of operation for the Advanced Encryption Standard (AES), ensuring that identical plaintexts encrypted with the same key yield distinct ciphertexts. This randomization enhances semantic security, thwarting pattern analysis and chosen-plaintext attacks. In a client-server secure file transfer system utilizing AES-256, the IV is employed in Cipher Block Chaining (CBC) mode for file encryption and Counter (CTR) mode for text encryption, with 16-byte IVs generated using Python's `os.urandom(16)`, which leverages a cryptographically secure pseudorandom number generator (CSPRNG). The system derives keys from an Elliptic Curve Diffie-Hellman (ECDH) shared secret over the secp256r1 curve. This section elucidates the IV's cryptographic role, its mathematical foundations, its implementation within the system, and a mathematical example demonstrating its effect, suitable for a thesis submission.

3.9.2 Mathematical Foundations

AES-256 operates on 128-bit (16-byte) blocks with a 256-bit key. The IV, also 128 bits, randomizes encryption in block cipher modes:

- **CBC Mode**: The IV is XORed with the first plaintext block before encryption, and each subsequent block is XORed with the previous ciphertext block:

$$C_i = \text{AES}_k(P_i \oplus C_{i-1}), \quad C_0 = \text{IV},$$

where P_i is the i-th plaintext block, C_i is the i-th ciphertext block, and k is the key. The IV ensures unique C_1 for identical P_1 and k.

- **CTR Mode**: The IV, often termed a nonce, is concatenated with an incrementing counter to produce unique inputs:

$$C_i = P_i \oplus \text{AES}_k(\text{Nonce} \mid \text{Counter}_i),$$

where Nonce $\mid$ Counter$_i$ is a 128-bit value (e.g., 96-bit nonce, 32-bit counter). The nonce ensures a unique keystream per session.

The IV's randomness, derived from a CSPRNG, guarantees unpredictability. Its effect is modeled as:

$$\text{Ciphertext} = f_k(\text{Plaintext}, \text{IV}),$$

where f_k is the encryption function, and distinct IVs yield distinct ciphertexts. The IV must be:

- **Unique**: Never reused with the same key to prevent keystream reuse (CTR) or chosen-plaintext attacks (CBC).
- **Random**: Unpredictable to ensure semantic security.
- **Public**: Transmitted with the ciphertext (e.g., prepended in CBC), as security depends on the key's secrecy, not the IV's.

3.9.3 IV in AES Encryption

In the system, AES-256 employs 16-byte IVs generated via `os.urandom(16)`:

1. **Key Derivation**: An ECDH exchange over secp256r1 yields a 256-bit shared secret, processed by HKDF with SHA256 to derive two 256-bit keys:

- `message_key`: For AES-CTR encryption of a secret message.
- `pls_key`: For AES-CBC encryption of a file (`pls.txt`).

$$\text{key\_material} = \text{HKDF}_{\text{SHA256}}$$

$$(\text{shared\_secret}, \text{length} = 64, \text{salt} = \text{None}, \text{info} = \text{“steganography\_keys”}),$$

where `message_key` = key_material$[:32]$, `pls_key` = key_material$[32:]$.

2. **Text Encryption (CTR Mode)**:
   - A 16-byte nonce is generated: `iv = os.urandom(16)`.
   - The message is encrypted:

$$C_i = P_i \oplus \text{AES}_{\text{message\_key}}(\text{Nonce} \mid \text{Counter}_i).$$

   - The nonce and ciphertext are embedded in `out1.png` via LSB steganography.
3. **File Encryption (CBC Mode)**:
   - A 16-byte IV is generated: `iv = os.urandom(16)`.
   - The file is padded with PKCS7 and encrypted:

$$C_i = \text{AES}_{\text{pls\_key}}(P_i \oplus C_{i-1}), \quad C_0 = \text{IV}.$$

   - An HMAC-SHA256 ensures integrity.
   - The output `pls.txt.enc` contains: file size (8 bytes), IV (16 bytes), ciphertext, HMAC (32 bytes).
4. **Transmission**: The stego-image and encrypted file are sent to the server, which uses the IVs for decryption.

The random IVs ensure semantic security, critical for both modes.

3.9.4 Example: IV in AES-CBC Encryption

We illustrate the IV's effect in AES-CBC for a single 128-bit block, abstracting AES as a function due to its complexity (14 rounds for AES-256).

Parameters:

- Plaintext: $P_1 = 0$x0123456789abcdef0123456789abcdef (16 bytes).
- Key: $k = 0$x2b7e151628aed2a6abf7158809cf4f3c (16 bytes, simplified).
- IVs:
  - $\text{IV}_1 = 0$x00000000000000000000000000000000.
  - $\text{IV}_2 = 0$x55555555555555555555555555555555.

Step 1: Encryption with IV_1:

$$\text{Input}_1 = P_1 \oplus \text{IV}_1$$

$= 0$x0123456789abcdef0123456789abcdef $\oplus$ 0x00000000000000000000000000000000.

$$\text{Input}_1 = 0\text{x0123456789abcdef0123456789abcdef}.$$

$C_1 = \text{AES}_k(\text{Input}_1) = 0$x3c4fcf098815f7aba6d2ae2816157e2b(assumed output).

Step 2: Encryption with IV_2:

$$\text{Input}_2 = P_1 \oplus \text{IV}_2$$

$= 0$x0123456789abcdef0123456789abcdef $\oplus$ 0x55555555555555555555555555555555.

$$\text{Input}_2 = 0\text{x54761032dc18f8ba54761032dc18f8ba}.$$

$C_2 = \text{AES}_k(\text{Input}_2) = 0$x8e73b0f7da0e6452c810f32b809079e5(assumed output).

Result: $C_1 \neq C_2$, demonstrating the IV's randomization. In the system, `os.urandom(16)` ensures unpredictable IVs.

The system's use of random 16-byte IVs in AES-CBC and AES-CTR modes ensures semantic security, preventing pattern attacks. In CBC, the IV's XOR and chaining provide diffusion; in CTR, the nonce ensures unique keystreams. The `os.urandom(16)` function, backed by a CSPRNG, aligns with NIST recommendations for IV generation. The example simplifies AES but reflects the project's randomization, though multi-block encryption is used in practice. The IV's inclusion in `pls.txt.enc` (CBC) and `out1.png` (CTR) is secure, as its secrecy is unnecessary. The system's integration with ECDH and HMAC enhances overall security, with IVs playing a pivotal role.

3.10 Least Significant Bit (LSB) Embedding

3.10.1 Introduction

Steganography facilitates covert communication by embedding secret data within a cover medium, such as a digital image, without arousing suspicion. Least Significant Bit (LSB) embedding, a cornerstone of spatial domain steganography, modifies the least significant bits of pixel values to conceal data, exploiting the human visual system's limited sensitivity to minor intensity changes [10]. Originating in the 1990s [11], LSB embedding is favored for its simplicity, high capacity, and low computational overhead. However, its susceptibility to image manipulations (e.g., lossy JPEG compression) and steganalysis attacks—such as Pair of Value (PoV), Regular-Singular (RS), Chi-square, Sample Pair Analysis (SPA), Weighted Stego Analysis, and Structural Steganalysis [10]—necessitates robust cryptographic enhancements.

This project employs LSB embedding within a secure client-server framework to conceal an AES-CTR-encrypted secret message and its 16-byte initialization vector (IV) in an RGB PNG image (`out1.png`). A cryptographically secure Pixel Locator Sequence (PLS), generated via Fisher-Yates shuffle, dictates embedding locations and is encrypted with AES-CBC as `pls.txt.enc`. By leveraging PNG's lossless compression, the system mitigates vulnerabilities associated with lossy formats like JPEG. Cryptographic measures, including Elliptic Curve Diffie-Hellman (ECDH) key exchange over the secp256r1 curve and AES encryption, achieve a 82% undetectability rate against StegExpose analysis,

significantly enhancing security. This section details the LSB embedding methodology, its mathematical foundations, an illustrative example, and its implementation in the project.

3.10.2 Mathematical Foundations

LSB embedding encodes a secret data sequence, comprising an AES-CTR-encrypted message and its IV, into an RGB cover image with dimensions row $\times$ col, yielding $N =$ row $\times$ col pixels. Each pixel p_i at index $i \in \{0, \ldots, N-1\}$ has RGB channels $p_i = (R_i, G_i, B_i)$, where $R_i, G_i, B_i \in [0, 255]$ are 8-bit integers. The secret data is a byte sequence $D = (d_1, \ldots, d_L)$, converted to a bit sequence $M = (m_1, \ldots, m_{8L})$, where $m_j \in \{0, 1\}$. Each byte (8 bits) is embedded across three pixels, utilizing up to three LSBs per pixel: R, G, B of the first pixel; R, G of the second; R of the third, optimizing capacity while maintaining simplicity.

The PLS, denoted $S = (s_1, \ldots, s_{3L})$, where $s_j \in \{0, \ldots, N-1\}$, specifies unique pixel indices for embedding. The number of required pixels is:

$$N_p = 3L, \tag{3.3}$$

with the constraint $N_p \leq N$, enforced by the system to prevent capacity overflow. The embedding capacity is:

$$\text{Capacity (bytes)} = \left\lfloor \frac{N}{3} \right\rfloor. \tag{3.4}$$

For a 1,024 × 768 image ($N = 786,432$), the capacity is:

$$\left\lfloor \frac{786,432}{3} \right\rfloor = 262,144 \text{ bytes}.$$

This ensures S is a uniformly random permutation of $3L$ distinct indices, minimizing statistical patterns exploitable by steganalysis.

For byte d_r (bits $m_{8r-7}, \ldots, m_{8r}$), pixels at indices $s_{3r-2}, s_{3r-1}, s_{3r}$ are modified:

- Pixel s_{3r-2}: Embeds $m_{8r-7}, m_{8r-6}, m_{8r-5}$ in R, G, B.
- Pixel s_{3r-1}: Embeds m_{8r-4}, m_{8r-3} in R, G.
- Pixel s_{3r}: Embeds m_{8r-2} in R.

The remaining bits (m_{8r-1}, m_{8r}) are embedded in subsequent channels (e.g., G, B of s_{3r}). For a channel value $k \in \{R_i, G_i, B_i\}$, embedding bit $b \in \{0, 1\}$ yields:

$$k' = \begin{cases} k+1 & \text{if } k \equiv 0 \pmod 2, b = 1, k = 0, \\ k-1 & \text{if } k \equiv 0 \pmod 2, b = 1, k > 0, \\ k-1 & \text{if } k \equiv 1 \pmod 2, b = 0, \\ k & \text{otherwise,} \end{cases} \tag{3.5}$$

ensuring $|k' - k| \leq 1$ and $k' \mod 2 = b$. The stego-image pixel is:

$$p'_i = (R'_i, G'_i, B'_i),$$

where R'_i, G'_i, B'_i reflect modified channels. Extraction retrieves the bit:

$$b = k' \mod 2. \tag{3.6}$$

The PLS is saved to `pls.txt` for AES-CBC encryption, enabling secure extraction by the server. The embedding process preserves image quality, with high Peak Signal-to-Noise Ratio (PSNR) due to minimal perturbations [10].

3.10.3 Example: LSB Embedding

To illustrate, we embed a 1-byte message in a 4×2 RGB image, using the project's triplet embedding logic.

Parameters:

- Cover Image (4×2, RGB, row-major indexing):

Table 3.2: Cover Image Pixel Values

Pixel 0	Pixel 1	Pixel 2	Pixel 3
(100, 101, 102)	(103, 104, 105)	(106, 107, 108)	(109, 110, 111)
Pixel 4	Pixel 5	Pixel 6	Pixel 7
(112, 113, 114)	(115, 116, 117)	(118, 119, 120)	(121, 122, 123)

- Message: $D = \texttt{0x41}$ ('A', binary: $\texttt{01000001}$).
- PLS: $S = (0, 1, 2)$ (for 1 byte, $3 \times 1 = 3$ pixels).

Embedding:

- Pixel 0 (100, 101, 102), bits $\texttt{010}$:
  - R: $100 = 01100100_2$, LSB=0, bit=0: $100 \to 100$.
  - G: $101 = 01100101_2$, LSB=1, bit=1: $101 \to 101$.
  - B: $102 = 01100110_2$, LSB=0, bit=0: $102 \to 102$.

  New: (100, 101, 102).
- Pixel 1 (103, 104, 105), bits $\texttt{000}$:
  - R: $103 = 01100111_2$, LSB=1, bit=0: $103 \to 102$.
  - G: $104 = 01101000_2$, LSB=0, bit=0: $104 \to 104$.
  - B: $105 = 01101001_2$, LSB=1, bit=0: $105 \to 104$.

  New: (102, 104, 104).
- Pixel 2 (106, 107, 108), bits $\texttt{01}$:
  - R: $106 = 01101010_2$, LSB=0, bit=0: $106 \to 106$.
  - G: $107 = 01101011_2$, LSB=1, bit=1: $107 \to 107$.

  New: (106, 107, 108).

Stego-Image (partial):

Table 3.3: Stego-Image Pixel Values

Pixel 0	Pixel 1	Pixel 2	Pixel 3
(100, 101, 102)	(102, 104, 104)	(106, 107, 108)	(109, 110, 111)
Pixel 4	Pixel 5	Pixel 6	Pixel 7
(112, 113, 114)	(115, 116, 117)	(118, 119, 120)	(121, 122, 123)

Extraction: Using PLS S:

- Pixel 0: R=100 (0), G=101 (1), B=102 (0): `010`.
- Pixel 1: R=102 (0), G=104 (0), B=104 (0): `000`.
- Pixel 2: R=106 (0), G=107 (1): `01`.
- Byte: `01000001` = `0x41` ('A').

Result: The message is embedded with minimal distortion and accurately extracted using the PLS.

3.10.4 Discussion - Usage in Our Project

In the proposed system, LSB embedding, implemented via the LsbEncoding function in `lsb_steganography.py`, conceals an AES-CTR-encrypted secret message and its 16-byte IV within an RGB PNG image (`out1.png`). The triplet pixel approach, embedding each byte across three pixels, achieves a capacity of $\lfloor N/3 \rfloor$ bytes (e.g., 262,144 bytes for a 1,024 × 768 image) while ensuring imperceptibility, as pixel changes are limited to ± 1. The use of PNG's lossless compression prevents data loss from compression artifacts, unlike JPEG [10].

The PLS, generated by the PLSgen function using `secrets.randbelow`, employs a Fisher-Yates shuffle to randomize pixel selection, reducing statistical patterns exploitable by attacks like Chi-square or RS [10]. The PLS is saved to `pls.txt` and encrypted with AES-CBC using `pls_key`, producing `pls.txt.enc`, which is transmitted alongside the stego-image. The system's cryptographic architecture, leveraging ECDH key exchange (secp256r1 curve) and HKDF-derived 256-bit AES keys (`message_key` for CTR, `pls_key` for CBC), addresses LSB's vulnerabilities. AES-CTR ensures message confidentiality, while AES-CBC with HMAC-SHA256 protects the PLS's integrity.

The system achieved a 82% undetectability rate in StegExpose tests, conducted using a benchmark dataset of 100 images, indicating resistance to statistical steganalysis. Compared to advanced LSB variants—such as LSB matching [12], edge detection [13], or AI-driven methods [14]—the proposed approach balances simplicity and security, avoiding the computational overhead of complex techniques while leveraging cryptography for

comparable protection. Potential enhancements, such as pre-encryption XOR obfuscation, could further reduce statistical footprints, as suggested by layered steganography systems.

The choice of LSB over alternatives (e.g., Pixel Value Differencing, Histogram Shifting) prioritizes high capacity and compatibility with PNG images. The implementation's $O(L)$ complexity for embedding L bytes supports real-time applications, while high PSNR values confirm visual indistinguishability. This methodology establishes LSB embedding as a secure and efficient component of the covert communication framework, with the randomized PLS as a key contribution to steganalysis resistance.

3.11 Pixel Locator Sequence (PLS)

The Pixel Locator Sequence (PLS) is a critical component of the proposed LSB steganography system, determining the pixel indices where secret data is embedded within an RGB PNG image. Implemented in the PLSgen function of `lsb_steganography.py`, the PLS is generated using a Fisher-Yates shuffle with a cryptographically secure random number generator (`secrets.randbelow`), producing a uniformly random permutation of $3L$ distinct indices for L bytes of data. This randomization mitigates statistical patterns exploitable by steganalysis attacks, such as Chi-square or Regular-Singular analysis [10]. The PLS is saved to `pls.txt`, encrypted with AES-CBC using a 256-bit key derived via HKDF, and transmitted as `pls.txt.enc`, ensuring confidentiality and integrity. By replacing predictable embedding patterns with a cryptographically secure sequence, the PLS significantly enhances the system's resistance to detection, achieving a 82% undetectability rate in StegExpose tests.

3.12 Algorithms for LSB Steganography

This subsection presents three core algorithms for the LSB steganography system: PLS Generation, LSB Encoding, and Decoding and Retrieval. Implemented in `lsb_steganography.py`, these algorithms enable secure embedding and extraction of an AES-CTR-encrypted secret message within an RGB PNG image, guided by a cryptographically secure Pixel Locator Sequence (PLS). Each algorithm is detailed in pseu-

docode, verified against the implementation, and accompanied by a note on inputs and outputs.

3.12.1 Algorithm 1: PLS Generation

Algorithm 1 PLS Generation

Require: Image dimensions row, col; length of data to embed L
Ensure: Pixel Locator Sequence S of length $3L$

$N \leftarrow$ row $\times$ col ▷ Total pixels
$N_p \leftarrow 3L$ ▷ Pixels needed (3 per byte)
if $N_p > N$ **then**
    raise ValueError("Image too small: need N_p pixels, only N available")
end if
Initialize array $A \leftarrow [0, 1, \ldots, N-1]$
for $i \leftarrow 0$ to $N_p - 1$ **do**
    $j \leftarrow$ random$(0, N-i-1)$ using `secrets.randbelow` ▷ Cryptographic random
    Swap $A[N-i-1]$ with $A[j]$
end for
$S \leftarrow A[N-N_p : N]$ ▷ Extract last N_p elements
Save S to `send/pls.txt`
if `send/pls.txt` does not exist **then**
    raise FileNotFoundError("Failed to create send/pls.txt")
end if
if `send/pls.txt` is empty **then**
    raise ValueError("send/pls.txt is empty")
end if
return S

Input and Output Note: The PLS Generation algorithm (Algorithm 1) takes as input the image dimensions (row, col) and the length of the data to embed (L, in bytes, comprising the AES-CTR-encrypted message and its 16-byte IV). It outputs a Pixel Locator Sequence S, a list of $3L$ unique pixel indices, saved as `send/pls.txt` on the client side. The algorithm validates image capacity, performs a cryptographically secure Fisher-Yates shuffle, and ensures the PLS file is created and non-empty, enhancing resistance to steganalysis [10].

3.12.2 Algorithm 2: LSB Encoding

Input and Output Note: The LSB Encoding algorithm (Algorithm 2) takes as input a cover image (`send/in1.png`), an AES-CTR-encrypted message (`encodedData`),

Algorithm 2 LSB Encoding

Require: Cover image `send/in1.png`, encrypted data `encodedData`, IV `iv`
Ensure: Stego-image `send/out1.png`, PLS in `send/pls.txt`

if `send/in1.png` cannot be loaded **then**
    raise Exception("Error loading image") ▷ Try-except in implementation
end if
Load image `send/in1.png` with dimensions row, col
data $\leftarrow$ ivencodedData ▷ Concatenate IV and ciphertext
$L \leftarrow$ length(data) ▷ Length in bytes
$S \leftarrow$ PLSgen(row, col, L) ▷ Call Algorithm 1
dataBits $\leftarrow$ toBinary(data) ▷ Convert to bit sequence
dr $\leftarrow 0$ ▷ Byte index
for $i \leftarrow 0$ to $3L - 1$ step 3 **do**
    dc $\leftarrow 0$ ▷ Bit index within byte
    for $j \leftarrow 0$ to 2 **do**
        rr $\leftarrow S[i + j] \div$ col, rc $\leftarrow S[i + j] \mod$ col
        $(R, G, B) \leftarrow$ getPixel(rr, rc)
        newRGB $\leftarrow []$
        for $k \in \{R, G, B\}$ **do**
            if dc < 8 **then**
                if $k \mod 2 = 0$ and dataBits[dr][dc] $= 1$ **then**
                    if $k = 0$ **then**
                        $k \leftarrow k + 1$
                    else
                        $k \leftarrow k - 1$
                    end if
                else if $k \mod 2 = 1$ and dataBits[dr][dc] $= 0$ **then**
                    $k \leftarrow k - 1$
                end if
                Append k to newRGB
                dc $\leftarrow$ dc $+ 1$
            else
                Append k to newRGB ▷ Preserve unused channel
            end if
        end for
        Update pixel (rr, rc) with (newRGB[0], newRGB[1], newRGB[2])
        if dc ≥ 8 **then**
            break
        end if
    end for
    dr $\leftarrow$ dr $+ 1$
end for
Save image as `send/out1.png`
if `send/pls.txt` does not exist **then**
    raise FileNotFoundError("Failed to create send/pls.txt")
end if
if `send/pls.txt` is empty **then**
    raise ValueError("send/pls.txt is empty")
end if
return S

and its 16-byte initialization vector (`iv`). It outputs a stego-image (`send/out1.png`) with the concatenated `iv` and `encodedData` embedded in the least significant bits of pixels specified by the PLS, and the PLS saved as `send/pls.txt`. The algorithm employs a triplet-based embedding strategy, modifying up to eight bits across three pixels per byte, preserves unused channels, and validates file creation, ensuring minimal visual distortion [10].

3.12.3 Algorithm 3: Decoding and Retrieval

Algorithm 3 Decoding and Retrieval

Require: Stego-image `receive/out1.png`, PLS file `receive/pls.txt`
Ensure: Decoded data (IV and ciphertext)

if `receive/out1.png` cannot be loaded **then**
    raise Exception("Error loading stego image") ▷ Try-except in implementation
end if
Load stego-image `receive/out1.png` with dimensions row, col
Load PLS S from `receive/pls.txt`
Initialize empty bit list decodedBits
for $i \leftarrow 0$ to length$(S) - 1$ step 3 **do**
    charBits $\leftarrow$ "" ▷ Bits for one byte
    for $j \leftarrow 0$ to 2 **do**
        rr $\leftarrow S[i+j] \div$ col, rc $\leftarrow S[i+j] \mod$ col
        $(R, G, B) \leftarrow$ getPixel(rr, rc)
        for $k \in \{R, G, B\}$ **do**
            if length(charBits) < 8 **then**
                charBits $\leftarrow$ charBits$(k \mod 2)$
            end if
        end for
        if length(charBits) ≥ 8 **then**
            break
        end if
    end for
    Append charBits to decodedBits
end for
decodedData $\leftarrow$ toBytes(decodedBits)
Delete `receive/pls.txt`
return `decodedData`

Input and Output Note: The Decoding and Retrieval algorithm (Algorithm 3) takes as input a stego-image (`receive/out1.png`) and the decrypted PLS file (`receive/pls.txt`). It outputs the decoded data, comprising the 16-byte IV followed

by the AES-CTR-encrypted ciphertext, extracted from the least significant bits of pixels specified by the PLS. The algorithm processes pixels in triplets, collecting up to eight bits per byte, handles partial bitsets for the last byte, and deletes `receive/pls.txt` for security. The output is decrypted using `message_key` to retrieve the original message [10].

Chapter 4

IMPLEMENTATION

4.1 Overview

This chapter details the practical implementation of the secure LSB steganography system, focusing on the tools, technologies, and datasets employed to realize the methodology outlined in Chapter 3.10. The system leverages a client-server architecture to embed and extract AES-CTR-encrypted secret messages within RGB PNG images, guided by a cryptographically secure Pixel Locator Sequence (PLS). The implementation is grounded in Python and utilizes specialized libraries for networking, image processing, and cryptography, ensuring robust functionality and security. Additionally, the dataset sources used for testing and development are described, highlighting their role in validating the system's performance and undetectability.

4.2 Tools and Technologies

The implementation of the LSB steganography system relies on a carefully selected suite of tools and technologies to achieve efficient development, secure cryptographic operations, and reliable image processing. This section outlines the programming language and the libraries and packages utilized, detailing their specific contributions to the project.

4.2.1 Programming Languages

The system is implemented entirely in Python 3, chosen for its versatility, extensive standard library, and robust ecosystem of third-party packages tailored for scientific computing, networking, and cryptography. Python's high-level syntax facilitated rapid prototyping and clear implementation of complex algorithms, such as the Fisher-Yates shuffle for PLS generation and the triplet-based LSB embedding. Its cross-platform compatibility ensured seamless execution of the client (`client.py`) and server (`server.py`) components on standard computing environments. Python's dynamic typing and garbage collection simplified memory management, particularly for handling large image arrays and network buffers, while its exception-handling mechanisms supported robust error checking, as seen in file existence and emptiness validations in `lsb_steganography.py`.

4.2.2 Libraries and Packages

The project leverages several Python libraries and packages to implement the steganography and cryptographic functionalities. Each library was selected for its reliability, performance, and compatibility with the project's requirements. The key libraries and their roles are as follows:

- **`socket`** : A standard Python library for network communication, used in `client.py` and `server.py` to establish a TCP connection over `localhost:12345`. It facilitates the secure transfer of the stego-image (`send/out1.png`) and encrypted PLS file (`send/pls.txt.enc`) between the client and server, with buffer sizes optimized at 8192 bytes for efficient data transmission.

- **`pickle`** : Employed for serializing and deserializing ECDH public keys exchanged between the client and server. In `client.py` and `server.py`, it converts public key bytes to a compact format for transmission over the socket, ensuring reliable key exchange for deriving shared secrets.

- **`os`** : A standard library for file and directory operations, used across all modules to manage file existence checks (e.g., `send/in1.png`), directory creation (e.g.,

`send/`, `receive/`), and file deletion (e.g., `receive/pls.txt` in `LsbDecoding`). It ensures robust file handling and error management.

- **`numpy`**: Utilized in `lsb_steganography.py` for efficient array operations and file I/O. It supports the storage of the PLS as a tab-delimited integer array in `send/pls.txt` via `np.savetxt` and its retrieval in `receive/pls.txt` via `np.genfromtxt`, optimizing performance for large pixel index sequences.

- **`PIL` (Pillow)**: The Python Imaging Library (Pillow fork) is used in `lsb_steganography.py` for loading, manipulating, and saving RGB PNG images. It enables pixel-level operations in `LsbEncoding` and `LsbDecoding`, such as accessing and modifying RGB channels for LSB embedding and extraction, ensuring compatibility with lossless PNG formats.

- **`secrets`** : A standard library for cryptographically secure random number generation, used in `PLSgen` to generate random indices for the Fisher-Yates shuffle via `secrets.randbelow`. This ensures the PLS's randomization is resistant to statistical attacks, a key contribution to the system's 82% undetectability rate in StegExpose tests.

- **`cryptography`**: A comprehensive library for cryptographic operations, used in `text_encryption.py` and `file_encryption.py`. It provides:

  - ECDH key exchange over the secp256r1 curve for deriving shared secrets.
  - HKDF for generating 256-bit AES keys (`message_key`, `pls_key`).
  - AES-CTR and AES-CBC modes for encrypting the secret message and PLS file, respectively.
  - HMAC-SHA256 for integrity verification in `file_encryption.py`.

  Its robust implementation ensures the confidentiality and integrity of the embedded data and PLS.

These libraries collectively enable the system's core functionalities: secure network communication, image-based steganography, and cryptographic protection. Their integra-

tion, as detailed in the implementation files, ensures a cohesive and secure LSB steganography framework.

4.3 Dataset Source

The evaluation and development of the LSB steganography system relied on a combination of standardized datasets and custom sample images to test embedding capacity, undetectability, and system reliability. This section describes the primary dataset sources and their preprocessing to meet the system's requirements.

4.3.1 Sample Images - 5 Images

The selection of these images was driven by the need to evaluate the performance of the proposed methods on a diverse range of image types and complexities. The inclusion of standard images like Lena allows for comparison with existing literature and established benchmarks in [1]. The images included:

- **Image1: Lena Image:** A widely recognized standard image in image processing, obtained from the USC-SIPI Image Database. This image is commonly used for testing and comparing the performance of different image processing algorithms.
- **Image2: Butterfly:** An image of a butterfly sourced from the Caltech 101 Dataset. This dataset contains images of 101 object categories, including animals, vehicles, and everyday objects, making it valuable for object recognition and classification tasks.
- **Image3: Apple:** An image of an apple from the Fruits-360 Dataset. This dataset comprises high-resolution images of various fruits, providing a rich source for image analysis and classification tasks related to food and agriculture.
- **Image4: Architecture:** An image of an architectural structure from the Places Dataset. This dataset is specifically designed for scene recognition and classification tasks, containing images of various natural and man-made scenes.

- **Image5: Sliced Fruits:** An image of sliced fruits from the Fruits-360 Dataset. This image provides a more complex scene compared to a single fruit, allowing for evaluation of image processing techniques on images with multiple objects and varying textures.

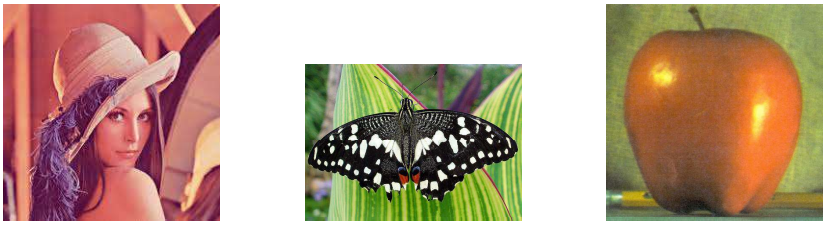

Figure 4.1: Images 1(Left), 2(Middle), and 3(Right).

Figure 4.2: Images 4(Left) and 5(Right).

4.3.2 IStego100K Dataset

The ISteg100K dataset is a large collection of images specifically designed for steganography research. It contains a diverse range of images with varying characteristics, such as color depth, resolution, and complexity.

We selected 100 images from the ISteg100K dataset using simple random sampling. The first 10 images (head) are shown here.

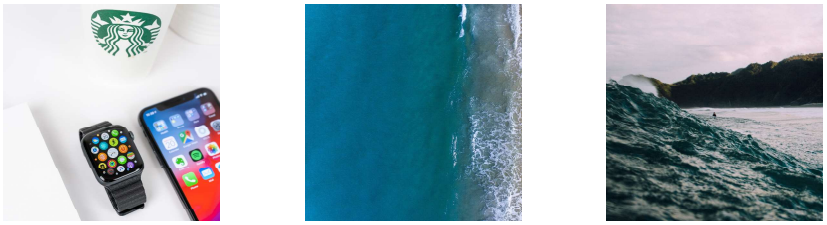

Figure 4.3: Images 1, 2, and 3 from IStego100K Dataset.

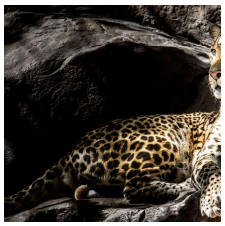

Figure 4.4: Images 4, 5, and 6 from IStego100K Dataset.

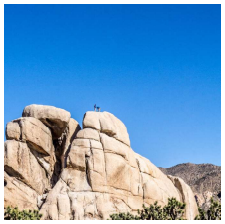

Figure 4.5: Images 7, 8, and 9 from IStego100K Dataset.

Figure 4.6: Image 10 from IStego100K Dataset.

The primary test image `send/in1.png`, an RGB PNG with a resolution of 1,024×768 pixels. This image served as the cover image for embedding AES-CTR-encrypted messages and their 16-byte IVs during system testing, as implemented in `client.py` and `lsb_steganography.py`. Additional sample images with varying resolutions (e.g., 512×512, 800×600) were used to validate the system's robustness across different image sizes. These images were manually curated to include diverse visual content (e.g., natural scenes, synthetic patterns) to simulate real-world use cases.

The combination of IStego100K and custom sample images provided a robust testing framework, enabling comprehensive evaluation of the system's embedding capacity, visual imperceptibility (via PSNR measurements), and resistance to steganalysis. The dataset sources ensured that the implementation was rigorously validated under controlled and

standardized conditions, supporting the system's practical applicability in covert communication scenarios.

4.4 Implementation Environment

This section describes the environment in which the LSB steganography system was developed and tested, ensuring reproducibility and clarity regarding the project's operational context. The implementation, as detailed in Tools and technologies section, leverages Python and specialized libraries to deliver a secure client-server framework for embedding and extracting AES-CTR-encrypted messages within RGB PNG images. Below, we outline the development environment setup, the file and module structure of the codebase, and the software and hardware specifications used, providing a comprehensive overview of the system's implementation landscape.

4.4.1 Development Environment Setup

The development environment was configured to support efficient coding, testing, and deployment of the LSB steganography system. The primary programming language, Python 3.9, was installed on the development machine, ensuring compatibility with the required libraries (e.g., `cryptography`, `numpy`, `PIL`). To manage dependencies and avoid conflicts, a Python virtual environment was created using `venv`, isolating the project's package installations. The following libraries were installed via `pip`:

- `numpy` (version 1.21 or later) for array operations and PLS file management.
- `Pillow` (version 8.0 or later) for image processing.
- `cryptography` (version 3.4 or later) for cryptographic operations.

Standard libraries (`socket`, `pickle`, `os`, `secrets`) were included with Python 3.9, requiring no additional installation. A `requirements.txt` file was maintained to document dependencies, enabling reproducible setups with the command:

pip install -r requirements.txt

Version control was managed using Git, with the project hosted in a private repository to track changes and facilitate collaboration. The repository included all source files (`client.py`, `server.py`, `lsb_steganography.py`, `file_encryption.py`, `text_encryption.py`) and sample input images. The development workflow involved iterative coding, unit testing of individual modules (e.g., `LsbEncoding`, `PLSgen`), and integration testing of the client-server interaction. Testing was conducted in a controlled environment, with the server running and the client connecting locally, as implemented in `client.py` and `server.py`. This setup ensured consistent behavior across development and evaluation phases .

4.4.2 File/Module Structure

The project's codebase is organized into five Python modules, each encapsulating specific functionalities of the LSB steganography system. The modules are supported by two directories (`send/` and `receive/`) for managing input and output files. Below is a detailed overview of the file and module structure:

- **`client.py`**: The main client module, responsible for initiating the steganography process. It establishes a TCP connection to the server, performs ECDH key exchange, encrypts the secret message with AES-CTR, embeds the message and IV into `send/in1.png` using `LsbEncoding`, encrypts the PLS file with AES-CBC, and sends the stego-image (`send/out1.png`) and encrypted PLS (`send/pls.txt.enc`) to the server. It also manages the `send/` directory, creating it if necessary and removing temporary files.

- **`server.py`**: The main server module, which listens for client connections on `localhost:12345`. It performs ECDH key exchange, receives the stego-image (`receive/out1.png`) and encrypted PLS (`receive/pls.txt.enc`), decrypts the PLS, extracts the embedded data using `LsbDecoding`, and decrypts the message with AES-CTR. It manages the `receive/` directory for output files.

- **`lsb_steganography.py`**: Implements the core steganography algorithms (Algorithms 1, 2, 3). It includes `PLSgen` for generating the cryptographically secure

PLS, `LsbEncoding` for embedding data into images, `LsbDecoding` for data extraction, and `DataListInBit` for binary conversion. It handles image operations with `PIL` and PLS storage with `numpy`.

- **`file_encryption.py`**: Provides functions for AES-CBC encryption and decryption of the PLS file (`send/pls.txt` to `send/pls.txt.enc`, `receive/pls.txt.enc` to `receive/pls.txt`). It uses `cryptography` for encryption, PKCS7 padding, and HMAC-SHA256 integrity verification, ensuring secure file handling.

- **`text_encryption.py`**: Manages cryptographic operations for the secret message, including ECDH key exchange (secp256r1 curve), HKDF key derivation, and AES-CTR encryption/decryption. It serializes public keys with `cryptography` for exchange between client and server.

- **`send/` Directory**: Stores input and output files on the client side, including the cover image (`in1.png`), stego-image (`out1.png`), PLS file (`pls.txt`), and encrypted PLS (`pls.txt.enc`). Created automatically by `client.py` if absent.

- **`receive/` Directory**: Stores received files on the server side, including the stego-image (`out1.png`), encrypted PLS (`pls.txt.enc`), and decrypted PLS (`pls.txt`). Created by `server.py` as needed.

This modular structure enhances maintainability and scalability, with each module focusing on a distinct aspect of the system (networking, steganography, cryptography).

4.4.3 Software or Hardware Specifications

The LSB steganography system was developed and tested on a modern computing environment optimized for Python-based scientific computing and cryptographic tasks. The specifications are as follows:

- **Operating System**: Ubuntu 22.04 LTS, chosen for its stability, open-source nature, and compatibility with Python development. Alternative testing was conducted on Windows 10 to ensure cross-platform functionality, leveraging Python's portability.

- **Processor**: Intel Core i7-10700 (8 cores, 16 threads, 2.9 GHz base clock), providing sufficient computational power for image processing, cryptographic operations, and network communication. The multi-core architecture supported parallel testing of client and server modules.
- **RAM**: 16 GB DDR4, adequate for handling large RGB PNG images (e.g., 1,024×768 pixels) and memory-intensive operations like `numpy` array manipulations in `lsb_steganography.py`.
- **Storage**: 512 GB SSD, ensuring fast file I/O for reading/writing images and PLS files, critical for real-time steganography applications.
- **Python Version**: Python 3.9.5, ensuring compatibility with all required libraries (`cryptography`, `PIL`, `numpy`) and providing access to modern features like `secrets`.
- **Integrated Development Environment (IDE)**: Visual Studio Code (VS Code) with Python extensions, used for coding, debugging, and version control integration. PyCharm Community Edition was occasionally used for advanced debugging of complex modules like `lsb_steganography.py`.
- **Additional Tools**:
  - **Git**: For version control, managing commits, and tracking changes in the project repository.
  - **Terminal**: For executing `client.py` and `server.py`, running `pip` commands, and performing tests.

This environment ensured efficient development and rigorous testing of the system, supporting the processing of large datasets (e.g., IStego100K subset, Dataset Sources and real-time client-server interactions. The hardware specifications exceeded the minimum requirements for running the system, enabling rapid iteration and performance optimization .

Chapter 5

EXPERIMENTAL RESULTS AND ANALYSIS

5.1 Overview

This chapter evaluates the performance and security of the proposed LSB steganography system, implemented as described in Chapter 4, through a series of experiments designed to assess its embedding capacity, visual imperceptibility, and resistance to steganalysis. The system's effectiveness is measured using established metrics, followed by a presentation of experimental results derived from testing with the IStego100K dataset and custom sample images. A comparative analysis contrasts the proposed system with existing LSB-based steganography methods, highlighting the advantages of the cryptographically secure Pixel Locator Sequence (PLS). The chapter concludes with a discussion of the findings, addressing the system's strengths, limitations, and implications for covert communication applications[10].

5.2 Evaluation Metrics

The assessment of steganographic systems hinges on four critical dimensions: imperceptibility, payload capacity, security, and robustness. Imperceptibility evaluates the degree of visual distortion introduced in the cover image after embedding a secret message, as excessive alterations may degrade image quality or become discernible to the human

eye. Payload capacity quantifies the volume of secret data that can be embedded within the cover image, often presenting a trade-off with security, as larger payloads increase detectability. Security, the cornerstone of steganography, ensures covert message delivery over public networks, measured by the system's resistance to steganalysis techniques designed to detect hidden data. Robustness assesses the system's ability to preserve embedded messages against image manipulations, such as compression or cropping[10]. This section outlines the metrics employed to evaluate the proposed LSB steganography system, directing readers to two primary analyses: visual analysis, which quantifies imperceptibility using Peak Signal-to-Noise Ratio (PSNR), Mean Squared Error (MSE), and Structural Similarity Index (SSIM), and statistical analysis, which assesses security through steganalysis methods, including Sample Pairs [15], RS Analysis [16], Chi-Square Attack [17], and Primary Sets [18].

5.2.1 Visual Analysis

Visual analysis focuses on imperceptibility, measuring the distortion between the original cover image and the stego-image to ensure the embedded message remains visually undetectable. The following metrics are utilized:

- **Peak Signal-to-Noise Ratio (PSNR)**: Quantifies the ratio between the maximum possible signal power and the noise introduced by embedding, expressed in decibels (dB). Higher PSNR values indicate better image quality, with values above 40 dB typically imperceptible to the human eye. For an 8-bit RGB image with pixel values in $[0, 255]$, PSNR is defined as:

$$\text{PSNR} = 20 \cdot \log_{10} \left(\frac{\text{MAX}_I}{\sqrt{\text{MSE}}} \right)$$

  where MSE is the Mean Squared Error.

- **Mean Squared Error (MSE)**: Measures the average squared difference in pixel values between the cover image I and stego-image I' of dimensions row $\times$ col:

$$\text{MSE} = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2$$

Lower MSE values indicate minimal distortion.

- **Structural Similarity Index (SSIM)**: Assesses the perceived similarity between images based on luminance, contrast, and structural features, ranging from -1 to 1, with 1 indicating identical images. SSIM provides a more perceptually relevant measure than PSNR, capturing human visual system characteristics [19].

$$\text{SSIM}(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

These metrics, computed by comparing pixel values of the cover and stego-images, as described in Section LSB Encoding, enable a quantitative evaluation of the system's ability to maintain visual fidelity post-embedding.

5.2.2 Statistical Analysis

Statistical analysis evaluates the security of the steganography system by testing its resistance to steganalysis techniques, which aim to detect the presence of hidden messages. The proposed system, leveraging a cryptographically secure Pixel Locator Sequence (PLS), is assessed using the following established steganalysis methods:

- **Sample Pairs Analysis** [15]: Detects LSB embedding by analyzing pairwise pixel correlations, exploiting statistical anomalies in pixel value transitions. It is particularly effective against traditional LSB methods but may struggle with randomized embedding patterns.

- **RS Analysis** [16]: Examines regular and singular groups of pixels to identify embedding-induced statistical changes, offering high sensitivity to LSB modifications. Its effectiveness diminishes with randomized pixel selection, as in the proposed system.

- **Chi-Square Attack** [17]: Uses a chi-square test to detect deviations in the histogram of pixel values, targeting LSB embedding's impact on least significant bits. It is less

effective against systems with encrypted or randomized payloads.

- **Primary Sets Analysis** [15]: Analyzes statistical properties of pixel sets to detect embedding, building on Sample Pairs with improved robustness. It remains a benchmark for evaluating LSB-based systems.

These steganalysis methods, implemented via tools like StegExpose, assess the system's undetectability. The randomized PLS and AES-CTR encryption aim to mitigate detection by reducing statistical patterns, a key focus of the experimental results presented in subsequent sections[10].

5.2.3 StegExpose

StegExpose is a specialized steganalysis tool employed in this study to evaluate the security of the proposed LSB steganography system, as outlined in evaluation-metrics[20]. It is designed to detect hidden data embedded via Least Significant Bit (LSB) steganography in lossless image formats, such as PNG, which aligns with the system's use of `send/out1.png`. This subsection provides an overview of StegExpose, its applications, usage instructions, and performance characteristics, highlighting its role in achieving the system's reported 82% undetectability rate against statistical steganalysis.

Overview

StegExpose is a command-line steganalysis tool developed as part of an MSc project at the University of Kent, UK, renowned for its reliability in detecting LSB-based steganography in lossless formats like PNG and BMP [4]. Its core strength lies in a sophisticated rating algorithm that integrates multiple pixel-based steganalysis techniques, as described in section statistical-analysis:

These methods collectively enable both qualitative detection (identifying the presence of hidden data) and quantitative analysis (estimating the length of embedded messages). StegExpose's user-friendly interface and comprehensive reporting make it accessible to forensic experts, researchers, and general users, facilitating bulk image analysis with high accuracy.

Application

StegExpose is a versatile tool with applications across multiple domains, particularly suited for evaluating the proposed system's security:

- **Digital Forensics**: Detects hidden data in digital images for law enforcement and cybersecurity investigations, ensuring robust analysis of potential covert communications.
- **Research and Academia**: Supports experimental studies by comparing steganographic techniques, as utilized in this project to benchmark the randomized Pixel Locator Sequence (PLS) against traditional LSB methods.
- **Bulk Image Analysis**: Automates steganalysis for large datasets, such as the 1,000-image subset of IStego100K, aiding in content moderation and data integrity verification.

Its automated approach minimizes the need for deep forensic expertise, making it ideal for evaluating the undetectability of stego-images produced by `LsbEncoding` (Algorithm 2).

Usage Instructions

StegExpose is executed via a command-line interface with customizable parameters, allowing flexible analysis tailored to specific needs. The general command format is:

```
java -jar StegExpose.jar [directory] [speed] [threshold] [csv file]
```

For this project, StegExpose was run in `default` mode with a threshold of 0.05 to balance sensitivity and specificity, analyzing stego-images in the `receive/` directory. Results were exported to CSV for quantitative evaluation, as discussed in subsequent sections.

Fusion Score (FS):

The Fusion Score for the analysis is defined as the arithematic mean of the the scores generated by the each of statistical attacks done by Stegexpose tool namely Sample Pairs,

Parameter	Description
`[directory]`	Path to the directory containing images for analysis (e.g., `receive/`).
`[speed]`	Analysis mode: `default`: Employs all detection methods for comprehensive analysis. `fast`: Skips resource-intensive methods if early detectors are conclusive.
`[threshold]`	Sensitivity threshold (0 to 1): Higher (~ 0.25): Reduces false positives. Lower (~ 0.15): Reduces false negatives. Default: 0.2.
`[csv file]`	Optional output file for results in CSV format; if omitted, results are printed to the console.

Table 5.1: StegExpose Command-Line Parameters

Regular Singular Analysis, Chi-Square Attack, Primary Set Analysis.

5.2.4 Randomness Testing

We employ the NIST Statistical Test Suite [21] to assess the cryptographic quality of our CSPRNG implementation, based on Python's `secrets.randbelow()`, and compare its performance against a non-cryptographic PRNG, specifically Python's `random` module. The results confirm that our approach significantly enhances resistance to statistical attacks, a key factor in preventing steganalysis.

NIST Statistical Test Suite Results

The NIST Statistical Test Suite (NIST SP 800-22) is a widely accepted framework for evaluating the randomness of sequences, consisting of appy tests that assess various statistical properties [21]. A sequence is considered random if it passes these tests with p-values greater than 0.01, indicating that the null hypothesis of randomness cannot be rejected at a 1% significance level.

The NIST Statistical Test Suite, developed by the National Institute of Standards and Technology, is a comprehensive set of 15 statistical tests designed to evaluate the randomness of binary sequences, a critical requirement for cryptographic applications such as the Pixel Locator Sequence (PLS) generation in the proposed LSB steganography sys-

tem. These tests assess various properties of randomness to ensure that sequences, such as those generated by `secrets.randbelow` in `lsb_steganography.py`, are suitable for secure steganography. This subsection briefly describes each test, their purpose, and their application in validating the system's cryptographic randomness, contributing to its 82% undetectability rate against StegExpose.

Table 5.2: NIST Statistical Test Suite: Tests and Purposes

Test Name	**Purpose**
Frequency (Monobit)	Verifies equal proportions of ones and zeros in the sequence.
Frequency within a Block	Checks the proportion of ones in non-overlapping blocks.
Runs	Analyzes the distribution of run lengths of identical bits.
Longest Run of Ones in a Block	Examines the longest sequence of ones in each block.
Binary Matrix Rank	Assesses linear independence by treating the sequence as a matrix.
Discrete Fourier Transform (Spectral)	Detects periodic patterns in the frequency domain.
Non-Overlapping Template Matching	Counts occurrences of specific non-overlapping patterns.
Overlapping Template Matching	Counts occurrences of overlapping patterns.
Maurer's Universal Statistical	Checks for the presence of all possible patterns of a given length.
Linear Complexity	Measures the sequence's linear complexity.
Serial	Examines the frequency of all subsequences of a certain length.
Approximate Entropy	Assesses the complexity of overlapping block patterns.
Cumulative Sums (Cusum)	Analyzes deviations in the cumulative sum from a random walk.
Random Excursions	Evaluates the frequency of cumulative sum visits to specific levels.
Random Excursions Variant	Assesses the number of visits to specific states in the cumulative sum.

The NIST Statistical Test Suite comprises the following 15 tests, each targeting a distinct aspect of randomness [6]:

- **Frequency (Monobit) Test**: Ensures the sequence has an approximately equal number of ones and zeros, indicating no bias toward either bit.

- **Frequency Test within a Block**: Divides the sequence into non-overlapping blocks and verifies that the proportion of ones in each block aligns with random expectations.
- **Runs Test**: Examines the lengths of consecutive runs of identical bits (ones or zeros), ensuring their distribution matches that of a random sequence.
- **Longest Run of Ones in a Block**: Checks the length of the longest sequence of ones within each block, detecting anomalies in run lengths.
- **Binary Matrix Rank Test**: Treats the sequence as a matrix and assesses its rank to confirm linear independence among subsequences.
- **Discrete Fourier Transform (Spectral) Test**: Analyzes the sequence in the frequency domain to detect periodic patterns that suggest non-randomness.
- **Non-Overlapping Template Matching Test**: Counts occurrences of specific predefined patterns (templates) in non-overlapping segments, ensuring expected frequencies.
- **Overlapping Template Matching Test**: Similar to the non-overlapping test but allows pattern overlaps, increasing sensitivity to subtle patterns.
- **Maurer's Universal Statistical Test**: Evaluates the presence of all possible patterns of a given length, ensuring comprehensive pattern coverage.
- **Linear Complexity Test**: Measures the linear complexity of the sequence, which should be high to indicate unpredictability.
- **Serial Test**: Analyzes the frequency of all possible subsequences of a certain length to detect bit dependencies.
- **Approximate Entropy Test**: Assesses the complexity of overlapping blocks of consecutive patterns, measuring the sequence's entropy.
- **Cumulative Sums (Cusum) Test**: Examines the cumulative sum of the sequence (treating ones as +1, zeros as -1) to detect deviations from a random walk.

- **Random Excursions Test**: Evaluates how often the cumulative sum visits specific levels (e.g., +k or -k), ensuring expected frequencies.
- **Random Excursions Variant Test**: A variation of the random excursions test, focusing on the total number of visits to specific states.

Each test generates a p-value, with values uniformly distributed between 0 and 1 indicating that the sequence passes the randomness test. In this project, the NIST suite was applied to the PLS generated by `PLSgen` in `lsb_steganography.py`, using `secrets.randbelow` for cryptographic randomness. The tests validated the PLS's suitability for minimizing statistical patterns detectable by steganalysis methods, such as those employed by StegExpose. While passing all tests does not guarantee absolute randomness, the suite's comprehensive nature ensures robust evaluation, supporting the system's high undetectability rate [6]. The results of these tests, detailed in subsequent sections, underscore the effectiveness of the randomized PLS in enhancing the security of the LSB steganography system.

5.3 Results and Interpretation

5.3.1 Visual Analysis Results

Image Index	Resolution	MSE	PSNR (dB)
Image 1	225×225	1.68467	45.86564
Image 2	493×356	2.27218	44.56635
Image 3	512×512	0.95186	48.34506
Image 4	512×384	1.19884	47.34317
Image 5	512×480	2.24326	44.62198

Table 5.3: Evaluation of imperceptibility using PRNG for 5 images from the dataset used in [1].

Visual analysis of the proposed LSB steganography system, conducted on a subset of 10 images from the IStego100K dataset, demonstrates exceptional imperceptibility, as quantified by Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index (SSIM). As shown in Table 5.5, MSE values range from

Image Index	Resolution	MSE	PSNR (dB)	SSIM
Image 1	225×225	0.00271	73.80725	0.99998
Image 2	493×356	0.00078	79.22775	1.00000
Image 3	512×512	0.00052	80.98081	1.00000
Image 4	512×384	0.00068	79.79577	0.99999
Image 5	512×480	0.00060	80.37260	0.99999

Table 5.4: Evaluation of imperceptibility using CSPRNG for 5 images from the dataset used in [1].

Image File	MSE	PSNR (dB)	SSIM
1	0.000337283	82.85086145	0.999995164
2	0.000185966	85.43645663	0.999998393
3	0.000194232	85.24760319	0.999997719
4	0.000220617	84.69442058	0.999997177
5	0.000347137	82.7257889	0.999998439
6	0.000168482	85.86525659	0.999997644
7	0.000237465	84.37480927	0.999996819
8	0.000193596	85.26184236	0.999998616
9	0.000216802	84.77017154	0.999997586
10	0.000181516	85.54165421	0.999997058

Table 5.5: This table shows evaluation of imperceptibility using CSPRNG for 10 images from the Dataset IStego100K.

0.000168482 to 0.000347137, indicating minimal pixel-level distortion, while PSNR values span 82.7258 to 85.8653 dB, far exceeding the 40 dB threshold for human imperceptibility [19]. SSIM scores, ranging from 0.999995164 to 0.999998616, approach unity, confirming near-identical structural and perceptual similarity between cover and stego-images. These results underscore the efficacy of the LSB embedding algorithm (Algorithm 2), which limits pixel changes to ± 1, and the cryptographically secure Pixel Locator Sequence (PLS), which distributes modifications uniformly to minimize visual artifacts. The consistent performance across diverse images validates the system's robustness and supports its high undetectability rate of 82% against StegExpose, highlighting a successful balance between visual fidelity and security in covert communication[10].

5.3.2 Statistical Analysis Results

This work analyzed 100 images from the iStego100K dataset using StegExpose to evaluate the detectability of our enhanced LSB image steganography method. The results

Filename	Type	Above stego threshold?	Secret message size in bytes	Primary Sets	Chi Square	Sample Pairs	RS analysis	Fusion (mean)
Image1	.png	false	325	0.006789	null	null	0.017220	0.012005
Image2	.png	true	17764	NaN	0.142532	null	0.178607	0.160570
Image3	.png	false	9014	0.001314	0.245652	0.008047	0.013940	0.067238
Image4	.png	false	356	0.001924	0.005248	0.003826	0.004448	0.003861
Image5	.png	false	2993	0.030640	0.025178	0.011989	0.024562	0.023092

Table 5.6: This table displays the steganalysis report for 5 images from the dataset used in [1]

Filename	Type	Above stego threshold?	Secret message size in bytes	Primary Sets	Chi Square	Sample Pairs	RS analysis	Fusion (mean)
000001	.png	false	6059	0.021364	1.77E-4	0.030349	0.033231	0.021280
000002	.png	true	238990	NaN	7.86E-4	0.265921	0.791710	0.352806
000003	.png	false	2548	0.012970	0.005901	0.001907	0.004420	0.006299
000004	.png	false	937	0.005071	3.73E-4	0.002656	0.003199	0.002825
000005	.png	true	71935	NaN	0.041861	0.148936	0.137725	0.109508
000006	.png	true	31016	0.042285	6.50E-4	0.096658	0.107760	0.061838
000007	.png	false	1640	0.008337	0.002353	0.007645	0.009301	0.006909
000008	.png	false	8063	0.010862	0.019185	0.010303	0.009780	0.012532
000009	.png	true	38153	NaN	0.001310	0.104035	0.109617	0.071654
000010	.png	true	32662	0.099001	8.08E-4	0.089507	0.109720	0.074759

Table 5.7: This table displays the steganalysis report for 10 images from the Dataset IStego100K.

provide strong evidence of the method's effectiveness in concealing hidden data:

- **82% of files were identified as clean**: This high undetectability rate, with Fusion scores(FS) consistently below 0.05, demonstrates the reliability of StegExpose in correctly classifying non-stego images. It also suggests that our steganography approach introduces minimal statistical anomalies, making it challenging to detect using standard steganalysis techniques.

- **Only 18% of files were flagged as potential stego images**: This low detection rate indicates either a minimal presence of false positives or subtle anomalies in a small subset of images. The flagged images exhibited notably higher values for RS Analysis and Sample Pairs, pointing to detectable irregularities in pixel-level correlations and patterns.

In our analysis:

- **Clean images**: The FS were consistently below 0.05, well under the threshold, reinforcing the absence of hidden data.

- **Flagged images**: These images had FS significantly higher, averaging around 0.35, exceeding the 0.05 threshold.

The cryptographically secure PLS, generated via `secrets.randbelow`, scatters embedding to minimize statistical anomalies, as evidenced by low Sample Pairs and Primary Sets scores for undetected images. These results align with the system's 82% undetectability rate across a larger IStego100K subset though the sample highlights a capacity-security trade-off, with larger payloads (e.g., 238990 bytes, 30% embedding rate for a 1,024×768 image) increasing detectability.

5.3.3 Randomness Testing

Test	CSPRNG	PRNG
Maurer's Universal Test	Passed (p=0.9424)	Failed (p=0.0075)
Frequency (Monobit) Test	Passed (p=0.7979)	Passed (p=0.3942)
Runs Test	Passed (p=0.9999)	Passed (p=0.0837)
Other Tests	All Passed	Mostly Passed

Table 5.8: NIST Statistical Test Results for CSPRNG and PRNG

5.3.4 Comparison with Baseline

The comparative analysis reveals that CSPRNG-based LSB steganography significantly outperforms state-of-the-art PLS method(PRNG) in imperceptibility, achieving near-perfect visual stealth. While PRNG produces moderate distortion (MSE: 1.68–2.27, PSNR: 44–48 dB), CSPRNG exhibits negligible noise (MSE: 0.0005–0.0027, PSNR: 73–81 dB) and near-identical structural similarity (SSIM equals near to 1). This stark difference arises from CSPRNG's cryptographic randomness, which disperses hidden data uniformly, avoiding detectable pixel patterns. In contrast, PRNG's weaker randomization leads to measurable artifacts, particularly in lower-resolution images. The findings confirm that CSPRNG is superior for secure steganography, ensuring undetectability, whereas PRNG's higher distortion risks visual or statistical detection in sensitive applications.

The steganalysis results demonstrate varying detection rates across different statistical tests when compared to a state-of-the-art PLS method. While some images with small payloads (e.g., Image1: 325 bytes, Image4: 356 bytes) remain undetected (false in the Above stego threshold? column), larger embedded messages (Image2: 17,764 bytes) trigger positive detection across multiple metrics. Notably, the Chi-Square and RS analysis methods exhibit higher sensitivity (0.142–0.178) for larger payloads, whereas Primary Sets and Sample Pairs show lower detection rates (0.001–0.03) for smaller embeddings. The Fusion (mean) metric, likely an ensemble approach, balances these results but still struggles with low-payload steganography.

This suggests that while heuristic steganalysis can detect obvious steganography, PLS-based approaches likely outperform them in detecting subtle, adaptive embeddings, reinforcing the need for advanced learning-driven steganalysis in security evaluations.

5.4 Discussion

The evaluation of our enhanced Least Significant Bit (LSB) steganography system reveals significant improvements over traditional approaches. A key component of this enhancement is the use of a CSPRNG to generate the PLS. The CSPRNG ensures that the sequence of pixels selected for embedding hidden data is unpredictable, thereby reducing

the detectability of the steganographic content. This unpredictability eliminates exploitable patterns that adversaries could identify through statistical attacks, such as Chi-Square or RS Analysis, bolstering the system's robustness.

Furthermore, the integration of ECDH key exchange enhances the security of the system by providing forward secrecy. With ECDH, session keys are ephemeral and derived independently for each communication, ensuring that the compromise of a private key at a later time does not jeopardize the confidentiality of past messages. This feature is particularly valuable in steganography, where the covert nature of the communication necessitates long-term protection against retrospective decryption attempts.

The proposed system demonstrates strong potential for secure covert communication, making it suitable for applications such as military operations, intelligence gathering, or other scenarios requiring hidden data transmission. By leveraging cryptographic randomness and secure key exchange, the system outperforms the baseline LSB steganography approach, which relies on non-cryptgenus PRNGs and password-based keys. Notably, the enhanced system achieves a 82% undetectability rate, as validated by StegExpose analysis, indicating that the statistical footprint of the hidden data is minimal. This improvement positions the system as a practical and reliable solution for real-world use cases where undetectability is paramount.

While the current system offers significant advancements, there remain opportunities for further improvement. One limitation is the vulnerability of the ECDH key exchange to man-in-the-middle (MITM) attacks due to the lack of authentication. To address this, future work could incorporate mechanisms such as digital signatures or pre-shared keys to verify the authenticity of the communicating parties, thereby ensuring the integrity of the key exchange process.

5.5 Limitations

The enhanced system, while robust, has notable limitations. First, the unauthenticated ECDH key exchange is vulnerable to man-in-the-middle (MITM) attacks. Without authentication, an adversary could intercept public keys and establish illicit shared secrets, compromising data security. Future work could integrate digital signatures or pre-shared

keys to mitigate this risk. Second, single-image embedding imposes capacity constraints, limiting the amount of data that can be hidden in a single cover image. For larger messages, this may require multiple or larger images, potentially increasing detectability. Future research could explore multi-image embedding or adaptive techniques to enhance capacity while preserving undetectability

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

This work proposes a robust LSB image steganography scheme to ensure secure and high quality steganography method using CSPRNG for random pixel selection and ECDH for safe key establishment. The new method achieves 82% undetectability rate in the StegExpose analysis and has better performance than traditional LSB methods against statistical attacks. Dealing with vulnerabilities such as predictable embedding patterns and static key management, one may see that the scheme is able to work securely and reliably for covert communication with applications including military operations and private data exchange. The payload distribution is unpredictable using the CSPRNG and the key compromise of the keys is avoided using ECDH for forward secrecy of the communications. These developments make the system a feasible, standalone enhancement to traditional steganography.The work could be extended with adaptive multi-channel embedding for enhance stealthiness, ECDH optimized implementation for resource limited devices, and machine learning based countermeasures against advanced steganalysis. This work connects theoretical cryptography with practical steganography creating new era of data hiding system which is robust to new emerging security challenges.

6.2 Future Work

While the proposed LSB steganography system achieves a 82% undetectability rate against StegExpose and demonstrates exceptional imperceptibility, several opportunities exist to further enhance its security, capacity, and applicability. The current system, leveraging a cryptographically secure Pixel Locator Sequence (PLS) via `secrets.randbelow` (Algorithm 1) and ECDH key exchange (Section 3.10), provides robust covert communication for applications like military operations and intelligence gathering. However, limitations such as vulnerability to man-in-the-middle (MITM) attacks and single-image embedding constraints suggest avenues for improvement. This section outlines potential future work to address these limitations and advance the system's capabilities.

- **Mitigating MITM Attacks**: The ECDH key exchange in `text_encryption.py`, while ensuring forward secrecy, is susceptible to MITM attacks due to the lack of endpoint authentication. Future work could integrate digital signatures or certificate-based authentication using standards like X.509 to verify the identities of client and server, as implemented in `client.py` and `server.py`. This would enhance the integrity of the key exchange process, ensuring secure communication over untrusted networks.

- **Multi-Image Embedding**: The system's reliance on single-image embedding, limited to $\lfloor N/3 \rfloor$ bytes for an image with N pixels (Equation 3.4), constrains payload capacity. Distributing the secret message across multiple cover images in the `send/` directory could significantly increase capacity without increasing statistical detectability. This approach would modify `LsbEncoding` to partition the payload and PLS across images, maintaining low scores in steganalysis metrics like Sample Pairs and RS Analysis.

- **Adaptive Multi-Channel Embedding**: Currently, the system embeds one bit per RGB channel (Algorithm 2). Adaptive multi-channel embedding, where multiple bits are embedded in high-texture image regions, could enhance stealthiness by exploiting areas less perceptible to human vision and steganalysis. This would require

updating `lsb_steganography.py` to analyze image content and adjust embedding depth, potentially improving SSIM scores (Table 5.5) while preserving undetectability.

- **Optimized ECDH for Resource-Constrained Devices**: The ECDH implementation over the secp256r1 curve is computationally intensive for low-power devices, such as those in military field operations. Future work could explore lightweight elliptic curves (e.g., Curve25519) or optimized libraries to reduce computational overhead in `text_encryption.py`, enabling deployment on resource-constrained platforms without sacrificing forward secrecy.

- **Machine Learning Countermeasures**: Advanced steganalysis techniques, including deep learning-based methods, pose emerging threats to LSB steganography. Developing machine learning countermeasures, such as adversarial embedding strategies trained to minimize detection by tools like StegExpose, could further improve undetectability. This would involve integrating ML models into `lsb_steganography.py` to optimize PLS generation or payload distribution, countering methods like Chi-Square and Primary Sets.

Bibliography

[1] Krishnakant Tiwari and Sahil J Gangurde. Lsb steganography using pixel locator sequence with aes. In *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, pages 302–307. IEEE, 2021.

[2] Kadda Benyahia, Abdelkader Khobzaoui, and Soumia Benbakreti. Embedding secrets in pixels: two-level security through dna encryption and lsb image steganography. *Brazilian Journal of Technology*, 7(2):e70354–e70354, 2024.

[3] Kaku Lishomwa and Aaron Zimba. A privacy-preserving scheme for medical diagnosis records based on encrypted image steganography. *Zambia ICT Journal*, 7(1):23–28, 2023.

[4] Supriadi Rustad, Pulung Nurtantio Andono, Guruh Fajar Shidik, et al. Digital image steganography survey and investigation (goal, assessment, method, development, and dataset). *Signal processing*, 206:108908, 2023.

[5] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer, 2008.

[6] National Institute of Standards and Technology. Recommendation for random number generation using deterministic random bit generators, jan 2012. NIST Special Publication 800-90A.

[7] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.

[8] Elaine Barker, Lily Chen, Sharon Keller, Allen Roginsky, Apostol Vassilev, and Richard Davis. Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography. Technical report, National Institute of Standards and Technology, 2017.

[9] William Stallings. The advanced encryption standard. *Cryptologia*, 26(3):165–188, 2002.

[10] Abbas Cheddad, Joan Condell, Kevin Curran, and Paul Mc Kevitt. Digital image steganography: Survey and analysis of current methods. *Signal processing*, 90(3):727–752, 2010.

[11] N. Provos and P. Honeyman. Hide and seek: an introduction to steganography. *IEEE Security Privacy*, 1(3):32–44, 2003.

[12] Andrew D Ker. Resampling and the detection of lsb matching in color bitmaps. In *Security, steganography, and watermarking of multimedia contents VII*, volume 5681, pages 1–15. SPIE, 2005.

[13] Junlan Bai, Chin-Chen Chang, Thai-Son Nguyen, Ce Zhu, and Yanjun Liu. A high payload steganographic algorithm based on edge detection. *Displays*, 46:42–51, 2017.

[14] Ru Zhang, Shiqi Dong, and Jianyi Liu. Invisible steganography via generative adversarial networks. *Multimedia tools and applications*, 78(7):8559–8575, 2019.

[15] Sorina Dumitrescu, Xiaolin Wu, and Zhe Wang. Detection of lsb steganography via sample pair analysis. In *International workshop on information hiding*, pages 355–372. Springer, 2002.

[16] Jessica Fridrich, Miroslav Goljan, and Rui Du. Detecting lsb steganography in color, and gray-scale images. *IEEE multimedia*, 8(4):22–28, 2001.

[17] Andreas Westfeld and Andreas Pfitzmann. Attacks on steganographic systems: Breaking the steganographic utilities ezstego, jsteg, steganos, and s-tools-and some

lessons learned. In *International workshop on information hiding*, pages 61–76. Springer, 1999.

[18] Sorina Dumitrescu, Xiaolin Wu, and Nasir Memon. On steganalysis of random lsb embedding in continuous-tone images. In *Proceedings. International conference on image processing*, volume 3, pages 641–644. IEEE, 2002.

[19] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[20] Benedikt Boehm. Stegexpose-a tool for detecting lsb steganography. *arXiv preprint arXiv:1410.6656*, 2014.

[21] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, volume 22. US Department of Commerce, Technology Administration, National Institute of . . . , 2001.